# OpenVMS Alpha Internals

## SCHEDULING AND PROCESS CONTROL

RUTH E. GOLDENBERG
DENISE E. DUMAS
SARO SARAVANAN

VERSION 7.0

**OpenVMS Alpha Internals and Data Structures**

Scheduling and Process Control

**OpenVMS Alpha Internals and Data Structures**

Scheduling and Process Control

## Digital Press Editorial Board

Samuel H. Fuller, Chairman
Richard W. Beane
Donald Z. Harbert
William R. Hawe
Richard J. Hollingsworth
William Laing
Richard F. Lary
Alan G. Nemeth
Pauline Nist
Robert M. Supnik

# OpenVMS Alpha Internals and Data Structures

## Scheduling and Process Control

Ruth E. Goldenberg

Denise E. Dumas

Saro Saravanan

Digital Press

Boston Oxford Johannesburg Melbourne New Delhi Singapore

Copyright © 1997 by Digital Equipment Corporation.

All rights reserved.

Digital Press™ is an imprint of Butterworth-Heinemann

A member of the Reed Elsevier Group

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the copyright holder.

In recognition of the importance of preserving what has been written, Butterworth-Heinemann prints its books on acid-free paper whenever possible.

Cover designer: Hannus Design Associates
Interior designer: David Ford
Copy editor: Alice Cheyer
Composition: Sarah Lemaire, Windfall Software (Paul C. Anagnostopoulos, MaryEllen N. Oliver) using ZzTEX
Index: Sarah Lemaire, Rosemary Simpson, John Mann

Motif is a registered trademark of the Open Software Foundation, Inc. (OSF); Distributed Computing Environment is an OSF product; Portable Operating System Interface (POSIX) is an IEEE standard. CI, DDIF, DEC, DECdtm, DECmigrate, DECnet, DECserver, DECthreads, DECwindows, Digital, MSCP, OpenVMS, PALcode, PATHWORKS for OpenVMS, Polycenter HSM for OpenVMS, TMSCP, TURBOchannel, VAX, VAX/VMS, VMS, VMS RMS, and VMScluster are trademarks of the Digital Equipment Corporation.

ISBN 1-55558-156-0

The publisher offers discounts on bulk orders of this book. For information, please write:
Manager of Special Sales, Digital Press
Butterworth-Heinemann
313 Washington St.
Newton, MA 02158

Order number: EY-T852E-DP

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America.

To the memory of
**Peter F. Conklin**

# Contents

**Preface xi**

**1 Life of a Process 1**

1.1 Process, Image, Kernel Thread, and Job 1
1.2 Process Context 2
1.3 Process Life Cycle Stages 7

**2 Scheduling 9**

2.1 Scheduling Data Structures 10
2.2 Hardware Context 20
2.3 Kernel Thread Priority 27
2.4 Scheduling States 32
2.5 Capabilities and Affinity 41
2.6 Scheduling Dynamics 45
2.7 Rescheduling Interrupt 68
2.8 Class Scheduling 78
2.9 Relevant Source Modules 80

**3 Event Flags 81**

3.1 Event Flags 81
3.2 PCB and KTB Fields Related to Event Flags 86
3.3 Associating to a Common Event Flag Cluster 88
3.4 Dissociating from a Common Event Flag Cluster 89
3.5 Deleting a Common Event Flag Cluster 90
3.6 Waiting for an Event Flag 90
3.7 Setting an Event Flag 100
3.8 Reading and Clearing Event Flags 102
3.9 Correct Use of Event Flags 102
3.10 Example of Interprocess Synchronization Through Common Event Flags 104
3.11 Relevant Source Modules 105

**4 Lock Management 107**

4.1 Overview 107
4.2 Lock Management Data Structures 109
4.3 Lock Management System Services 126

4.4 Handling Deadlocks 141
4.5 Relevant Source Modules 149

**5 Kernel Threads 151**

5.1 Overview 151
5.2 Multithreading Controls 153
5.3 Kernel Thread Data Structures 153
5.4 Transitions to and from Multithreading 163
5.5 Kernel Thread Creation and Deletion 169
5.6 Scheduling 178
5.7 Implications of Shared Memory 182
5.8 Synchronization within a Multithreaded Process 183
5.9 Upcalls from the OpenVMS Executive to a User Mode Thread Manager 204
5.10 DECthreads Thread Manager Overview 211
5.11 Relevant Source Modules 221

**6 Process Control and Communication 223**

6.1 Requirements for Affecting Another Process 224
6.2 Process Information System Services 229
6.3 System Services Affecting Process Computability 243
6.4 Miscellaneous Process Attribute Changes 255
6.5 Interprocess Communication 259
6.6 Relevant Source Modules 261

**7 Process Creation 263**

7.1 Overview 263
7.2 Process Creation in the Context of the Creating Process 265
7.3 Process Identification 279
7.4 Swapper's Role in Process Creation 283
7.5 Process Creation in the Context of the New Process 290
7.6 Relevant Source Modules 297

**8 Image Activation and Exit 299**

8.1 Images 299
8.2 Image and Image-Related Data Structures 311
8.3 Image Activation 325
8.4 Address Relocations and Fixups 348
8.5 Image Startup 360
8.6 Image Exit 364

8.7 Image and Process Rundown 370
8.8 Process Privileges 379
8.9 Relevant Source Modules 382

**9 Translated Image Environment 383**

9.1 Overview 383
9.2 Emulating the VAX Environment 384
9.3 Translating an Image 391
9.4 Calling Standards and Jacketing 396
9.5 Translated Image Initiation 404
9.6 Execution within the Translated Environment 406
9.7 Relevant Source Modules 415

**10 Process Deletion 417**

10.1 Process Deletion in Context of Requesting Process 417
10.2 Process Deletion in Context of Process Being Deleted 421
10.3 Process Deletion in System Context 428
10.4 Deletion of a Process that Owns Subprocesses 430
10.5 Relevant Source Module 432

**11 Process Dynamics 433**

11.1 Overview 433
11.2 Single-Image Process 434
11.3 The Job Controller and Process Creation 439
11.4 The LOGINOUT Image 442
11.5 CLIs and Image Processing 452
11.6 SPAWN and ATTACH 461
11.7 Multiple-Image Processes 466
11.8 LOGINOUT's Other Roles 468
11.9 Relevant Source Modules 470

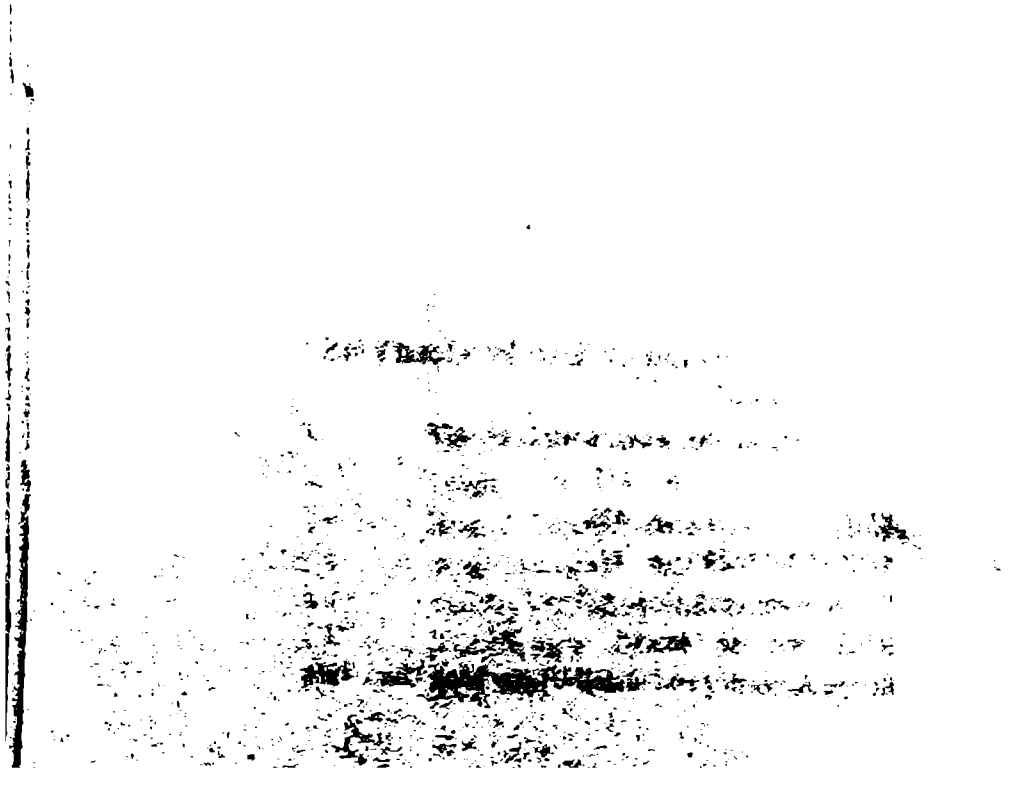
**App. A Lock and Resource Use by OpenVMS Components 471**

A.1 Aspects of Resource and Lock Use 471
A.2 Executive Lock Use 473
A.3 $MOUNT Lock Use 476
A.4 $DISMOU Lock Use 477
A.5 File System Lock Use 478
A.6 RMS Lock Use 485
A.7 Image Activator and Install Utility Lock Use 495

A.8 NETACP Lock Use 497
A.9 Job Controller and Queue Manager Lock Use 501
A.10 SYSMAN Lock Use 507

**Selected Acronyms 509**

**Index 519**



# Preface

This book describes the scheduling subsystem of OpenVMS Alpha Version 7.0 and the system services that create, control, and delete processes. It discusses in detail how images are mapped into a process's virtual address space and prepared for execution. It also explains OpenVMS Alpha support for multithreading, which enables multiple threads sharing a virtual address space to execute concurrently.

This book is an update to part of the book *OpenVMS AXP Internals and Data Structures: Version 1.5* (1994). It is not wholly independent of that book. For example, a typical chapter has many pointers to other chapters, some in this book and some in its predecessor, for expansion of topics mentioned only briefly in the originating chapter. References to chapters within this book are by number; for example, Chapter 2. References to chapters in *OpenVMS AXP Internals and Data Structures: Version 1.5* are by title; for example, Chapter *Synchronization Techniques*.

For conceptual background on internals topics not covered in this book, consult the Version 1.5 book. Although details such as data structure layouts will likely have changed since Version 1.5, much of the conceptual foundation of OpenVMS is unchanged. (The version numbers of successive OpenVMS Alpha releases were advanced directly from Version 1.5 to Versions 6.1 and 6.2 and then to Version 7.0, to indicate feature parity with OpenVMS VAX versions.)

## CONVENTIONS

A number of conventions are used throughout the text and figures of this volume.

During the life of the VAX VMS operating system, the exact form of its name has changed several times: from VAX/VMS Version 1.0 to VAX VMS Version 5.0 to OpenVMS VAX Version 5.5. In describing the evolution of VMS algorithms and discussing the foundation of the OpenVMS Alpha operating system, this book refers to the OpenVMS VAX operating system by whichever name is appropriate for the version referenced.

The term *executive* refers to those parts of the operating system that are loaded into and that execute from system space. The executive includes the system base images, SYS$BASE_IMAGE.EXE and SYS$PUBLIC_VECTORS.EXE, and a number of other loadable executive images. Because there is no need to distinguish different types of executive image, this book generally shortens the term *loadable executive image* to *executive image*.

The terms *system* and *OpenVMS system* describe the entire OpenVMS software package, including privileged processes, utilities, and other support

software as well as the executive itself. The OpenVMS system consists of many different components, each a different file. One component is the system base image, SYS$BASE_IMAGE.EXE. Others are executive images, device drivers, command language interpreters, and utility programs.

The source modules from which these components are built and their listings are divided into facilities. Each facility is a directory on a source or listing medium containing sources and command procedures to build one or more components. The facility [DRIVER], for example, contains sources for most of the device drivers. The facility [SYSBOOT] contains sources for the secondary bootstrap program, SYSBOOT. The facility [SYS] contains the sources that make up the base images and many executive images.

This book identifies a [SYS] facility source module only by its file name. It identifies a module from any other facility by facility directory name and file name. For example, [SYSGEN]SYSGEN refers to the source for the system generation utility (SYSGEN).

Almost all source modules are built so as to produce object modules and listing files of the same file name as the source module. Thus, a reference in this book to a source module name identifies the file name of the listing file as well. In a case where the two names differ, the text explicitly identifies the name of the listing file. Appendix *Use of Listing and Map Files* discusses how to locate a module in the source listings.

This book identifies a macro from SYS$LIBRARY:LIB.MLB by only its name, for instance, WFIKPCH. The macro library of all other macros is specified.

Ported from VAX VMS, many OpenVMS Alpha executive routines have JSB-type entry points. That is, they were originally written to be entered with a VAX JSB instruction rather than a VAX CALLS or CALLG instruction. Typically this was done for performance reasons at a time when most of the executive was written in VAX MACRO and the rest in VAX BLISS. On an Alpha CPU, however, there is little difference in the code generated for a MACRO-32 JSB instruction and for a MACRO-32 CALLS instruction.

As part of adding support for high-level language device drivers and other system code, a standard call-type entry point has been added for each JSB-type entry point. A call-type entry point has the string _STD in its name; for example, SCH$POSTEF and SCH_STD$POSTEF are the two entry points of the routine that sets event flags. New routines typically have only one entry point, a standard call-type entry point with a name that includes the string _STD.

In documenting code flow from one routine to another, this volume uses whichever name the code does. In documenting specific routines, for the most part, it uses the shorter JSB-type name, for example, SCH$POSTEF

rather than SCH_STD$POSTEF. In the index, entries for a routine appear under the shorter form of the name.

The unmodified terms *process control block* and *PCB* refer to the software data structure used by the scheduler. The data structure that contains a process's hardware context, the hardware privileged context block (HWPCB), is always called the HWPCB.

The term *inner access modes* means those access modes with more privilege. The term *outer access modes* means those with less privilege. Thus, the innermost access mode is kernel and the outermost mode is user.

The term *SYSGEN parameter* refers to a system cell that can be altered by a system manager to affect system operation. Traditionally, these parameters were altered through either the SYSGEN utility or SYSBOOT, the secondary bootstrap program. Although they can also now be altered through the SYSMAN utility or AUTOGEN command procedure, this volume continues to use the traditional term *SYSGEN parameter.* SYSGEN parameters include both the dynamic parameters, which can be changed on the running system, and the static parameters, whose changes do not take effect until the next system boot. These parameters are referred to by their parameter names rather than by the symbolic names of the global locations where their values are stored. Appendix *SYSGEN Parameters and Their Locations* relates parameter names to their corresponding global locations.

The terms *byte index, word index, longword index,* and *quadword index* derive from methods of VAX operand access that use context-indexed addressing modes. That is, the index value is multiplied by 1, 2, 4, or 8 (for bytes, words, longwords, or quadwords, respectively) as part of operand evaluation, to calculate the effective address of the operand. Although the Alpha architecture does not include these addressing modes, the concept of context indexing is relevant to various OpenVMS Alpha data structures and tables.

A term in small capital letters refers to the formal name of an argument to an OpenVMS system service, for example, the LOGNAM argument.

A bit field is sometimes described by its starting and ending bit numbers within angle brackets; for example, the interrupt priority level of the processor, in the processor status bits (12:8), is contained in bits 8 through 12.

Three conventions are observed for lists:

- In lists like this one, where no order or hierarchy exists, list elements are indicated by leading diamond-shaped bullets. Sublists without hierarchy are indicated by round bullets.
- Lists that indicate an ordered set of operations are numbered. Sublists that indicate an ordered set of operations are lettered.

- Numbered lists with the numbers enclosed in circles indicate a correspondence between the list elements and numbered items in a figure or example.

Several conventions are observed for figures. In all diagrams of memory, the lowest virtual address appears at the top of the page and addresses increase toward the bottom of the page. Thus, the direction of stack growth is depicted upward from the bottom of the page. In diagrams that display more detail, such as bytes within longwords, addresses increase from right to left. That is, the lowest addressed byte (or bit) in a longword is on the right-hand side of a figure and the most significant byte (or bit) is on the left-hand side.

Each field in a data structure layout is represented by a rectangle. In many figures, the rectangle contains the last part of the name of the field, excluding the structure name, data type designator, and leading underscore. A rectangle the full width of the diagram generally represents a longword regardless of its depth. A field smaller than a longword is represented in proportion to its size; for example, bytes and words are quarter- and half-width rectangles. A quadword is generally represented by a full-width rectangle with a short horizontal line segment midway down each side. In some figures, a rectangle the full width of the diagram represents a quadword. In these figures, bit position numbers above the top rectangle show numbers from 0 to 63 to indicate that the rectangle represents a quadword.

For example, Figure 5.2 shows the layout of the process control block (PCB) and kernel thread block (KTB). The rectangle labeled PRVCPU represents the longword PCB$L_PRVCPU; the rectangle labeled PGFLCHAR, the word PCB$W_PGFLCHAR; and the rectangle labeled PRVASN, the quadword PCB$Q_PRVASN.

In almost all data structure figures, the data structure's full-width rectangles represent longwords aligned on longword boundaries. In a few data structures, a horizontal row of boxes represents fields whose sizes do not total a longword. Without this practice, most of the fields in this kind of structure would be split into two part-width rectangles in adjoining rows, because they are unaligned longwords.

Some data structures have alternative definitions for fields or areas within them. A field with multiple names is represented by a box combining the names separated by slash (/) characters. An area with multiple layouts is shown as a rectangle with a dashed line separating the alternative definitions. For example, in Figure 5.2, fields PCB$L_FREWSLE_PARAM and PCB$L_PQB are two names for the same field. Figure 5.2 also shows an example of alternative definitions for an area; the longword at PCB$L_EFC2P is also divided into the word PCB$W_PGFLCHAR, byte PCB$B_PGFLINDEX, and an unused byte.

A data structure field containing the address of another data structure in the same figure is represented by a bullet connected to an arrow pointing to

the other structure. Where possible, the arrow points to the rightmost end of the field, that is, to bit 0. A field containing a value used as an index into that or another data structure is represented by an x connected to an arrow pointing to the indexed location.

Two conventions indicate elisions in a data structure layout. A specific amount of space is shown as a rectangle whose sides contain dots. Text within the rectangle indicates the amount of space it represents. Field PCB$T_LNAME in Figure 5.2, for example, represents 16 bytes.

An indeterminate amount of space, often unnamed, representing omitted and undescribed fields, is indicated by a rectangle whose sides are intersected by short parallel horizontal lines. For example, Figure 2.2, which identifies only the PCB fields related to scheduling, contains seven sets of omitted fields among the labeled fields.

In a typical figure that represents a code flow, such as Figure 2.13, time flows downward. Each different environment in which the code executes is represented by a column in the figure. The headings above the columns identify the environment characteristics, for example, "User Image" or "Kernel Mode".

A code flow figure represents only the events in the code most relevant to the current discussion. A description of code within a routine begins with the routine's name in bold-face type followed by text lines describing the routine's actions. When one routine calls another, the routine nesting is shown by indents. A lightning bolt represents an exception or interrupt. A diamond represents a branch test. An arrow indicates a transfer of control, typically from one routine to another.

## ACKNOWLEDGMENTS

In addition to acknowledging the work of the many contributors to *OpenVMS AXP Internals and Data Structures: Version 1.5*, we would like to acknowledge the contributors to this book.

Sarah Lemaire made significant and varied contributions to the creation of this volume. She created new art, painstakingly and skillfully applied the figure and text edits, and updated the index and the acronym list. We were very glad of her help.

A number of reviewers reviewed substantial portions of the book and made suggestions that improved its quality: Paul Benoit, Dave Bernardo, Ken Blaylock, Ian Brockbank, Richard Critz, Stu Davidson, Clair Grant, Mike Harvey, Bonnie Huval, Jim Kauffman, Andy Kühnel, Rob de Maat, Ben Thomas, and Elinor Woods.

We were delighted to work again with Alice Cheyer, who edited previous versions of this book. With her lively intelligence and meticulous editing, she brought consistency to a collection of chapters written in different voices.

Paul Anagnostopoulos converted our VAX DOCUMENT chapter files into TEX, translating David Ford's design into type with his original program, ZzTEX.

We are grateful to Steve Noyes, Group Engineering Manager for OpenVMS Engineering, for funding the creation of this updated series. We'd also like to thank our managers, Ron Higgins and Ken Munsell, for their support.

Ruth E. Goldenberg
Denise E. Dumas
July 1996

# 1 Life of a Process

The science of life . . .
is a superb and dazzlingly lighted hall
which may be reached only by passing through
a long and ghastly kitchen.
Claude Bernard, *Introduction à l'Étude de la Médecine Expérimentale*

This chapter introduces some basic OpenVMS concepts related to processes and summarizes the life of a process. Major events, such as creation, scheduling, preemption, and deletion, are described in more detail throughout the rest of this volume.

## 1.1 PROCESS, IMAGE, KERNEL THREAD, AND JOB

When a user logs into an OpenVMS system, the executive creates an environment called a process from which the user can enter commands and run programs. This environment includes a virtual address space and characteristics such as user name, account name, default disk, default directory, process quotas, and logical name definitions. It also includes a process security profile, consisting of user identification code (UIC), privileges, and identifiers. Logging into the system is a common, but not unique, way to create a process. Chapter 11 discusses other circumstances in which processes are created.

Many of the characteristics derive initially from the user's authorization information, specified by the system manager when the user was authorized to use the system. Others derive from SYSGEN parameters. Some characteristics can be altered by the user through command or program action.

The virtual address space is divided into regions: system regions that are shared by all processes and process-private regions that are unique to each process. Executive code and data occupy the system regions. Within the process's virtual address space, the user can run programs, which are also known as images.

An OpenVMS Alpha image is a file that contains binary data and instructions that can be executed on an OpenVMS Alpha system. The linker constructs an image from one or more object modules produced by a language processor such as a compiler. A user can initiate image execution, for example, through the Digital command language (DCL) RUN command. In response, an operating system component called the image activator reads the image and maps it into the process's address space. The executive reads

image pages into memory as they are referenced. The CPU executes the instructions within the image. Chapter 8 discusses the image activator, and Chapter *Paging Dynamics*, the paging mechanism.

A thread of execution is a sequential flow of instruction execution. A traditional program is single-threaded and consists of a mainline thread of execution. In contrast, a multithreaded program divides its work among multiple threads that can run concurrently. OpenVMS supplies DECthreads, a run-time library, to facilitate the creation, synchronization, and deletion of multiple user mode threads.

OpenVMS Alpha Version 7.0 supports multiple execution contexts within a process: each execution context has its own hardware context and stacks, and can execute independently of the others; all execution contexts within a process share the same virtual address space. The term *kernel thread* refers to one of these execution contexts and its current thread of execution. This volume uses the term *multithreaded* to refer to a process with multiple kernel threads.

Each process is created with one kernel thread and can create up to the maximum number specified by SYSGEN parameter MULTITHREAD. OpenVMS Alpha Version 7.0 schedules kernel threads, rather than processes, for execution. Even if one kernel thread in a process goes into a wait, other kernel threads in that process remain executable. Moreover, multiple kernel threads of the same process can execute concurrently on different CPUs in a symmetric multiprocessing (SMP) system.

With the assistance of DECthreads, a process can have many more user mode threads than kernel threads. If a user mode thread goes into a wait, DECthreads can schedule a different user mode thread in the kernel thread.

The term *kernel thread* is somewhat misleading, because instructions executing in a kernel thread can run in any of the four access modes, not only kernel mode. The term arose to distinguish the new multithreaded model, supported within the OpenVMS executive (or kernel), from the model of previous releases, supported only by DECthreads user mode run-time library code. Chapter 5 contains more information on kernel thread support.

A process can create subprocesses. A job is the collection of a top-level or master process, all the subprocesses created by it, and all the subprocesses created by them. The concept of a job exists in part for the purpose of sharing resources. Processes in the same job can also share jobwide logical names and I/O devices. They do not share process-private virtual address space.

## 1.2 PROCESS CONTEXT

The software and hardware contexts of a process and its kernel threads are described by a number of data structures in various regions of the process's virtual address space. Figure 1.1 shows some of the relations among these structures for a single-threaded process. The structures are introduced in the sections that follow.

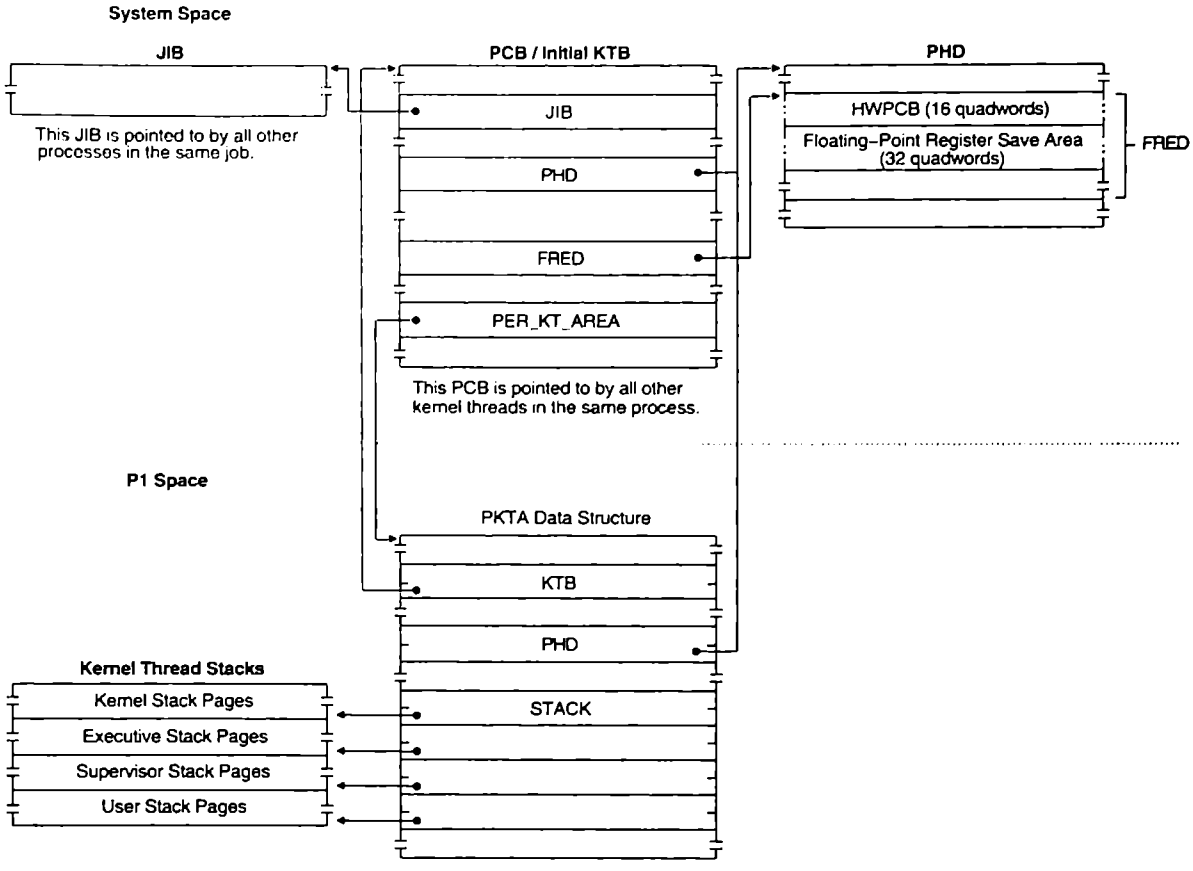



**Figure 1.1**
Data Structures That Describe Process Context

### 1.2.1 Hardware Context

When a kernel thread within a process is current, its hardware context consists of integer registers, the program counter (PC), the processor status (PS), and the process-specific processor registers: for example, the page table base register (PTBR), the asynchronous system trap enable (ASTEN) register, and the AST summary register (ASTSR). If the thread has performed floating-point arithmetic, its hardware context includes the floating-point registers.

A kernel thread can be executing in any of four access modes: user, supervisor, executive, and kernel. The PS identifies the thread's current access mode. Each kernel thread has four stacks and stack pointers, one for each of the access modes. The stack pointers are also part of hardware context. Code executing in a particular access mode runs on the stack associated with that access mode.

When a kernel thread is removed from execution, some of its nonprivileged hardware context (namely, the integer registers, PC, and PS) is saved on its kernel stack. A subset of its hardware context, called hardware privileged context, is stored in a data structure called a hardware privileged context block (HWPCB). Figure 2.8 shows the layout of the HWPCB. If the kernel

thread performed floating-point arithmetic, the contents of the floating-point registers are saved in the process's process header (PHD).

When the kernel thread is placed back into execution, its privileged context is restored from the HWPCB and its nonprivileged context from the kernel stack. If appropriate, floating-point registers are restored from the PHD.

The area in which the HWPCB and floating-point registers are saved is called a floating-point register and execution data (FRED) structure. By default the PHD can accommodate one FRED, for the initial kernel thread. When multithreading is initiated in a process, the PHD is expanded to accommodate FREDs for the maximum possible number of additional kernel threads. Figure 5.5 shows the layout of the FRED and the expanded PHD.

### 1.2.2 Virtual Address Space

OpenVMS Alpha virtual address space was originally based on the OpenVMS VAX model of address space and consisted of P0, P1, and system space. In OpenVMS Alpha Version 7.0, the definition of virtual address space has been expanded to include P2, S2, and page table (PT) space:

- Each process has a process-private P0 space to run programs.
- Each process has a process-private P1 space for its stacks, permanent process control information maintained by the operating system, and a command language interpreter (CLI) if one is being used.
- Each process has a process-private P2 space for data storage.
- Each process accesses a shared system space region called S0/S1 space, occupied by operating system components and systemwide data.
- Each process accesses an additional system space shared region called S2 space, occupied by operating system data.
- The virtual address space of a process is described by a hierarchy of page tables, which contain information used by the CPU to translate virtual addresses into physical addresses. Each process has a PT space for the page tables that map its own process-private and the shared system spaces.

  Each process's process-private page tables are unique to that process. Every process, however, maps system space using the same physical and virtual pages of PT space. The part of PT space that maps process-private spaces is itself process-private. The part of PT space that maps system space is shared.

  All kernel threads in a process access the same process-private virtual address space, using the same page tables.

Figure 1.2 summarizes these areas. Chapters *Memory Management Overview* and *Memory Management Data Structures* contain more information about the page table hierarchy, virtual address space, and address translation.

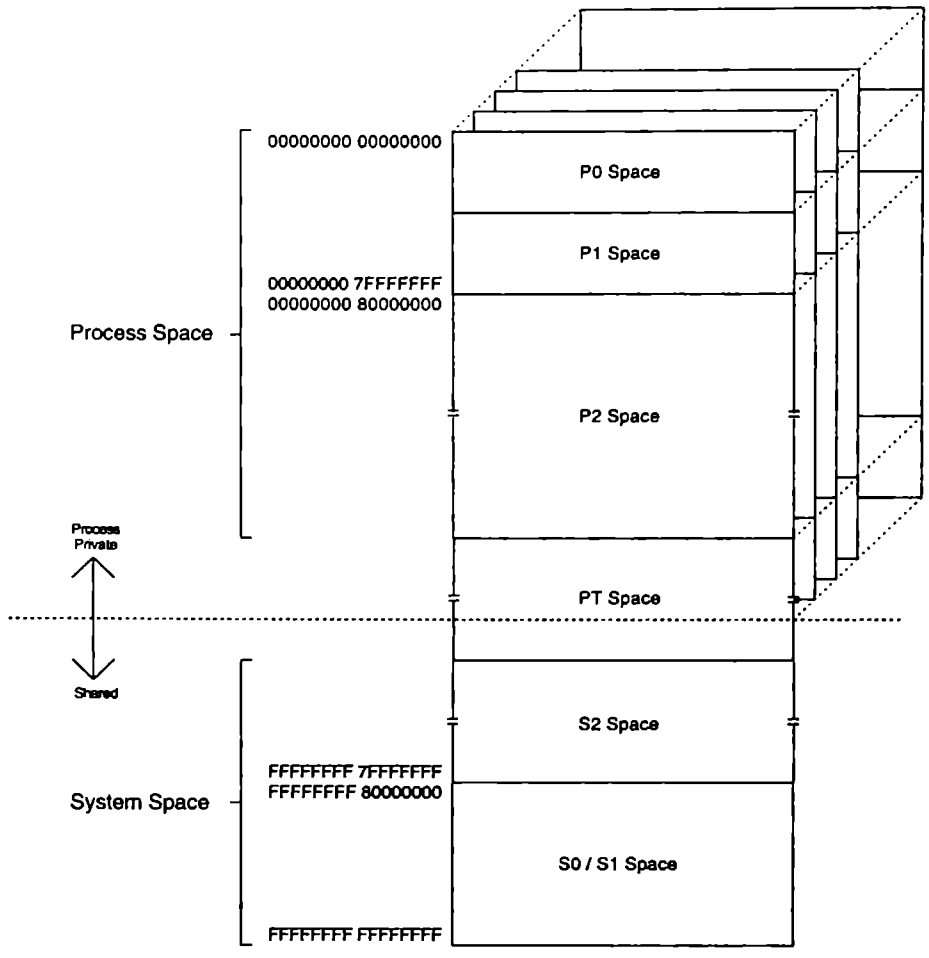



**Figure 1.2**
Virtual Address Space

### 1.2.3 Software Context

Software context consists of all the data required by various parts of the operating system to control that portion of common resources allocated to a given process or kernel thread. This context includes the kernel thread's software priority, its scheduling state, process quotas and limits, process page file assignments and reservations, and miscellaneous information, such as process name and identification.

**1.2.3.1 Process Control Block and Kernel Thread Block.** Process information that must always be in memory is stored in a data structure called the process control block (PCB). It contains information such as the process's identification (PID), name, processwide quotas and limits, and pointers to other process data structures. In general, process-specific information that might be accessed outside of process context is stored in its PCB. When a process is

outswapped, its location in a swap file on a mass storage device is recorded in its PCB.

Support for multithreaded processes requires separating processwide information from thread-specific data and replicating thread-specific data. Some PCB fields, such as software priority and scheduling state, have been moved to a thread-specific data structure called a kernel thread block (KTB). A KTB exists for each kernel thread the process creates.

A multithreaded process has an additional data structure that records the addresses of its kernel threads' KTBs. Chapter 5 discusses this structure in more detail.

To minimize changes to the PCB, the layout of the KTB was designed so that it could overlay the PCB. That is, each structure is sparse, with some fields designated PCB fields and others KTB fields. Newly defined fields specific to multithread support are at the end of the KTB. The overlay makes it possible for the PCB and the initial kernel thread's KTB to occupy the same memory.

Because many KTB fields retain their symbolic PCB names, this design minimizes changes to modules that reference fields in only the initial thread's KTB. Figure 5.2 shows the layout of the PCB and KTB.

1.2.3.2 **Job Information Block.** Jobwide quotas and limits are recorded in a data structure called the job information block (JIB). A JIB also has information about the user that is common to all processes in the job, such as user name, master process ID, and number of processes in the job.

The PCB of every process in a job contains a pointer to the common JIB. Figure 7.3 shows this structure.

1.2.3.3 **Process Header.** A PHD contains some of the process-specific information that is needed when the process is resident. It contains much of the process's memory management information. In particular, the process section table within it identifies the location and type of each section mapped in the process-private part of virtual address space. The swapper uses the working set list, a list of valid virtual pages currently in use by the process, when it removes the process from memory (outswaps) or brings the process back into memory (inswaps). Chapter *Memory Management Data Structures* describes the PHD and its substructures in more detail.

The PHD also contains some of the kernel thread's hardware process context (see Section 1.2.1).

1.2.3.4 **P1 Space.** Other process-specific information is stored in the P1 region of the process's virtual address space. This includes information about the image that is currently executing. Information stored in a process's P1 space is accessible only from the context of any of its kernel threads, because P1 space is process-private.

1.2.3.5 **Per Kernel Thread Area.** Each kernel thread has a private P1 space area, called the per kernel thread area (PKTA), which includes space for the thread's four stacks and some data. The data cells are organized into a data structure within the PKTA. Pages of P1 space for the stacks and the data associated with the initial kernel thread are created when the process is created. In a multithreaded process, when a new kernel thread is created, the P1 space for its PKTA is created.

Figure 5.6 shows this area in more detail.

## 1.3 PROCESS LIFE CYCLE STAGES

Every process, except the swapper process, is created as the result of a Create Process ($CREPRC) system service request from a kernel thread running in another process context. The system service request specifies what image is to run in the new process, what the process's initial input source is to be, and what its initial output destination is.

The $CREPRC system service, running in the context of the creator process, determines the quotas and limits of the new process and allocates and initializes the new process's PCB, KTB, and JIB. It triggers the swapper process to materialize a minimal hardware context for the new process. System space executive code running in the new process context further initializes the process's software and hardware contexts and its process-private virtual address space. Chapter 7 describes this sequence in detail.

Executive code then requests the Image Activator ($IMGACT) system service to map the image specified to run in that process. Chapter 8 describes the image activator and its use of memory management system services to create the virtual address space the image occupies.

Often the image that first runs in the new process is LOGINOUT, one of whose major functions is to activate a CLI image to run in the process. Unlike most images, the CLI is activated in P1 space, is mapped for the duration of the process, and runs in supervisor mode. It accepts commands from the process's input source and carries them out. The CLI's response to many commands is to request activation of the particular image associated with the command. Such an image is typically activated in P0 space.

When the image exits, its virtual address space is deleted, and control returns to the CLI, which accepts further commands. Chapter 11 describes LOGINOUT and the role of a CLI in passing control from one image to the next, as well as the system components that create most processes. Chapter 8 documents the Exit ($EXIT) system service.

By default a process has a single execution context or kernel thread. An image running in a process can initiate multithreading by requesting the $INIT_MULTITHREAD system service. It can then create additional kernel threads by requesting the $CREATE_KERNEL_THREAD system service. At image exit, the $END_MULTITHREAD system service, requested by the image or

the $EXIT system service, terminates any threads that still exist other than the process's initial kernel thread. Chapter 5 describes these services and other aspects of multithreading in detail.

Once created, a kernel thread can be selected to execute. Executing, it can request certain system services or incur page faults so as to be no longer computable. If DECthreads is running as a user mode thread manager in the kernel thread's process, the executive may notify DECthreads of the wait so that it can select another user mode thread to run in the execution context provided by the kernel thread.

If DECthreads is not running, executive scheduling code places the kernel thread into a wait state and selects the highest priority computable kernel thread to run. The waiting and newly current kernel threads can be from the same process or different processes. Chapter 2 includes information on the scheduler database, scheduling priorities, and scheduling dynamics.

Chapter 6 describes some of the services by which one process affects the computability of another process or kernel thread. Chapter 3 describes methods by which a process or kernel thread uses event flags to synchronize its execution with the completion of system operations such as I/O requests. Chapter 4 describes the use of OpenVMS resources and locks to synchronize access to entities such as global sections and files. Appendix A summarizes the lock and resource use by various OpenVMS components.

Most images that execute in an OpenVMS Alpha process contain only native Alpha instructions. Under the control of the DECmigrate tools, however, it is possible to execute an image originally built as an OpenVMS VAX image whose VAX instructions have been translated to Alpha instructions. Chapter 9 discusses the run-time environment in which such an image executes and OpenVMS support of that environment.

When an image exits in a process without a CLI, the process is deleted. In an interactive process with a CLI, when one image exits, the user can enter another command to cause activation of another image. If an image requests deletion of the process or the user requests deletion by entering the command LOGOUT, the process is deleted. Process deletion includes releasing resources the process had acquired and removing all traces of the process from the system. Chapter 10 describes how these steps are carried out.

# 2 Scheduling

> It is equally bad when one speeds on the guest unwilling to go, and when he holds back one who is hastening. Rather one should befriend the guest who is there, but speed him when he wishes.
>
> Homer, *The Odyssey*

In previous versions of OpenVMS, the unit of scheduling was the process. In OpenVMS Alpha Version 7.0, a process running a multithreaded application can have multiple kernel threads, each executing on a different CPU in a symmetric multiprocessing (SMP) environment. Consequently, the unit of scheduling is the kernel thread rather than the process. For scheduling purposes, there is little distinction between a single-threaded process and its initial kernel thread.

Only one kernel thread can run on a processor at once. Scheduling is the mechanism that selects a kernel thread to run.

The characteristics most relevant to the scheduling of a kernel thread are

- Kernel thread priority, which determines the precedence of kernel threads for execution
- Scheduling state, which defines the readiness of a kernel thread to be scheduled for execution, its computability or lack thereof
- CPU capability or affinity requirements, which constrain the set of processors of an SMP system on which a kernel thread can execute

Scheduling code running on a particular processor identifies and selects for execution the highest priority kernel thread that can execute on that processor and places it into execution. A kernel thread currently executing enters a wait state when it makes a direct or indirect request for a system operation that cannot complete immediately. A waiting kernel thread becomes computable as the result of system events, such as the setting of an event flag or the queuing of an asynchronous system trap (AST), and may preempt a current kernel thread.

This chapter first describes the data structures related to scheduling and the significance of priority, scheduling state, capabilities, and affinity. It then describes the dynamics of their interactions—how changes in one characteristic can affect the others and the mechanisms by which the characteristics change. Finally, it describes the rescheduling interrupt service routine in detail.

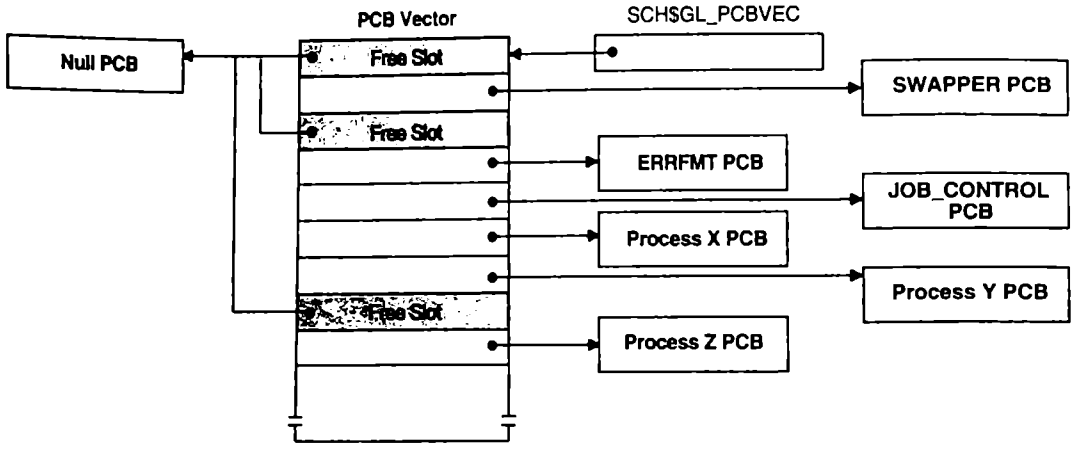



**Figure 2.1**
Sample PCB Vector

## 2.1 SCHEDULING DATA STRUCTURES

This section describes most of the system data structures related to scheduling.

Processwide information that must always be in memory is stored in a data structure called the process control block (PCB). It contains information such as the process's identification (PID), name, processwide quotas and limits, and pointers to other data structures. When a process is outswapped, its location in a swap file on a mass storage device is recorded in its PCB. Section 2.1.1 describes PCB fields related to scheduling.

The addresses of all PCBs in the system are recorded in the PCB vector, a longword array, which is allocated from nonpaged pool during system initialization. Its address is stored in SCH$GL_PCBVEC. It contains an entry for each possible process in the system. The number of entries in the array (and therefore the maximum number of processes allowed at any given time on a system) is determined by the SYSGEN parameter MAXPROCESSCNT.

Figure 2.1 shows an example of the contents of the PCB vector.

The first entry in the vector contains the address of the null PCB, a placeholder PCB. The second entry contains the address of the swapper process PCB. All other entries in the vector initially contain the address of the null PCB to show that they are unused. The term *slot* is sometimes used to refer to the entries in this vector. For example, the SHOW MEMORY command displays the total, free, resident, and swapped numbers of "process entry slots."

The null PCB is defined statically as a placeholder. In earlier versions of the OpenVMS operating system, the null PCB described a process called the null process. This process no longer exists, but there is still a need for a placeholder PCB so that each system pointer to a PCB can point to a valid PCB, even when there is no associated process.

The fundamental scheduling data structure is the kernel thread block (KTB). It specifies the scheduling state, priority, and capability and affinity requirements of a kernel thread, and records many other kernel thread characteristics. Section 2.1.2 describes KTB fields related to scheduling. Section 2.1.3 describes the scheduling state queues, the queues in which KTBs of kernel threads in the same scheduling states are linked.

The KTB was designed to overlay the PCB. That is, each structure is sparse, with some fields designated KTB fields and others PCB fields. In particular, the overlay makes it possible for the initial kernel thread's KTB to occupy the same memory as the PCB. For a single-threaded process, there is thus little distinction between the process's PCB and its initial kernel thread's KTB.

The KTB contains the address of the process's process header (PHD). The PHD contains much of the process's memory management information. It also contains some data related to scheduling (see Section 2.1.4). In particular, the PHD contains a substructure called a floating-point register and execution data structure (FRED) for each kernel thread in the process. A FRED includes additional kernel-thread-specific data related to scheduling and a hardware privileged context block (HWPCB).

An HWPCB is a structure in which privileged context is loaded and stored through the `CALL_PAL SWPCTX` instruction. Section 2.2.1 describes the HWPCB, and Section 2.1.4, the PHD and FRED scheduling fields.

The data structure called the per-CPU database supports SMP. The per-CPU database records processor-specific information. Each CPU has its own per-CPU database. Section 2.1.5 describes the fields in this structure related to scheduling.

Each CPU is identified by an ID, a number from 0 to 31. The system mask SCH$GL_IDLE_CPUS has a bit corresponding to each CPU. When set, the bit indicates that the CPU is idle and has no current kernel thread executing on it. The bit is cleared to indicate that the CPU should repeat its attempt to select a kernel thread to execute.

Several systemwide data structures related to kernel thread priority are described in Section 2.3.

The set of PCBs, KTBs, scheduling state queues, and related data structures is known as the scheduler database. The SCHED spinlock synchronizes access to the set of data structures, although not necessarily to all fields within each structure. Chapter *Synchronization Techniques* describes the implementation and use of spinlocks.

### 2.1.1 PCB Fields Related to Scheduling

When a process is created, a PCB is allocated for it from nonpaged pool. A process continues to use the same PCB until it is deleted and its PCB deallocated.

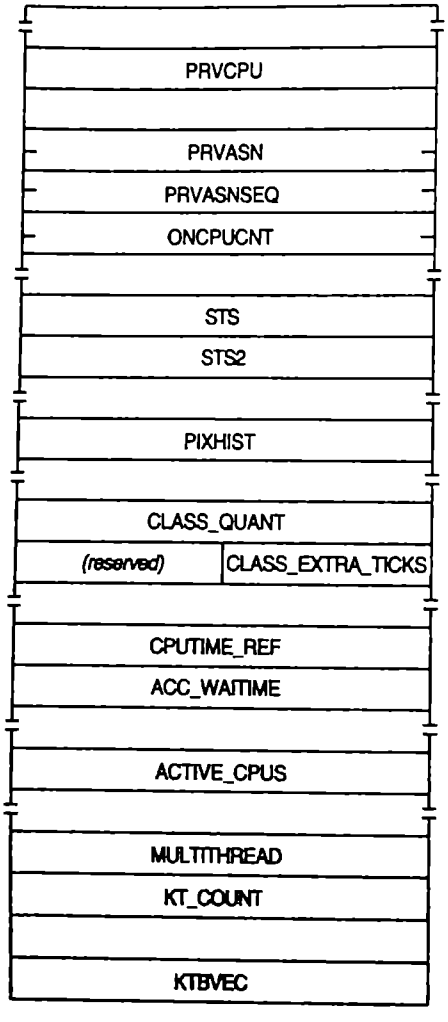



**Figure 2.2**
**Process Control Block (PCB) Fields Used in Scheduling**

Figure 2.2 illustrates the fields of the PCB that are particularly important for scheduling.

PCB$L_PRVCPU, PCB$Q_PRVASN, and PCB$Q_PRVASNSEQ are discussed in Section 2.2.3.

PCB$Q_ONCPUCNT contains the number of kernel threads in this process that are currently executing on a CPU. PCB$L_ACTIVE_CPUS contains a bit mask with a bit corresponding to each CPU in an SMP system. When set, the bit indicates that a kernel thread of this process is executing on a particular CPU.

PCB$L_MULTITHREAD specifies whether multithreading is active in this process. A value of 0 indicates no multithreading: the process only has its initial kernel thread. A nonzero value is the maximum number of kernel

threads the process can create. A value of 1 indicates that multithreading has been initiated; although only one kernel thread can be created, upcalls to a user mode thread manager are possible (see Chapter 5).

PCB$L_KT_COUNT contains the current number of kernel threads in this process. PCB$L_KTBVEC is the address of the array of KTB addresses for this process's kernel threads. Chapter 5 gives more information on multithreading.

PCB$L_STS and PCB$L_STS2 contain status bits relevant to the process as a whole, for example, PCB$V_DISAWS, which when set means that automatic working set list adjustment is disabled. The bit PCB$V_RES in PCB$L_STS is of particular significance to scheduling. When set, it indicates that the process is in memory rather than outswapped. Table 7.2 describes the flags in the status longwords.

PCB$L_CPUTIME_REF records the amount of CPU time the process had accumulated since the last time its ratio of wait time to compute time was calculated. This ratio is calculated to determine whether memory should be reclaimed from a process's working set. PCB$L_ACC_WAITIME records the amount of wait time the process has accumulated since then. Section 2.6.1.6 describes the use of these fields in more detail.

PCB$L_PIXHIST is described in Section 2.6.7.

PCB$L_CLASS_QUANT and PCB$W_CLASS_EXTRA_TICKS are described in Section 2.8.

### 2.1.2 KTB Fields Related to Scheduling

When a kernel thread is created, a KTB is allocated for it from nonpaged pool. A kernel thread continues to use the same KTB until it is deleted and its KTB deallocated.

Figure 2.3 illustrates the fields of the KTB that are particularly important for scheduling. Other KTB fields are described in other chapters, in particular, Chapters 3, 5, *ASTs*, and *Synchronization Techniques*.

KTB$W_SIZE and KTB$B_TYPE are the standard dynamic data structure fields.

The scheduling state of a kernel thread is specified by its KTB$L_STATE field. All kernel threads in the system are in the current (CUR) state, a wait state, computable resident (COM) state, or computable outswapped (COMO) state. Section 2.4 summarizes the scheduling states and the transitions among them.

The KTBs of kernel threads in most scheduling states are queued together with those of other kernel threads in the same state so that they can be located more easily by scheduling routines. The scheduling state queue link fields, KTB$L_SQFL and KTB$L_SQBL, link a KTB into a scheduling state queue. The various kernel thread state queues are described in Section 2.1.3.

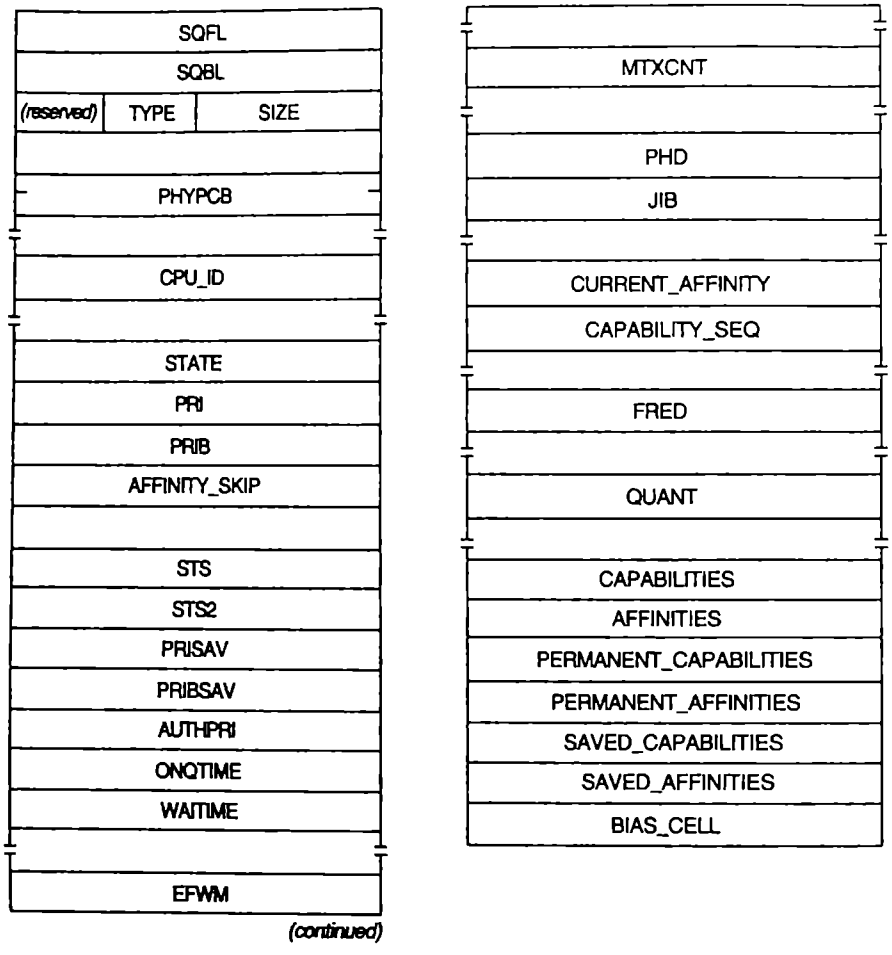



**Figure 2.3**
**Kernel Thread Block (KTB) Fields Used in Scheduling**

The physical address of a kernel thread's HWPCB is stored in KTB$Q_PHYPCB. The virtual address of a kernel thread's FRED is stored in KTB$L_FRED.

KTB$L_STS and KTB$L_STS2, the status longwords, contain various flags describing the status of the kernel thread. For example, KTB$V_WALL, in KTB$L_STS, when set, indicates that the kernel thread is waiting for all the event flags in the event flag wait mask to be set.

Several KTB fields are related to priority. Section 2.3 describes these fields.

When a kernel thread is in an event flag wait (see Section 2.4.3.1) or a miscellaneous wait state (see Section 2.4.3.3), KTB$L_EFWM identifies the flags or resource for which the process waits.

KTB$L_PHD contains the address of the PHD (see Section 2.1.4).

KTB$L_JIB contains the address of the job information block (JIB). All processes in a job tree share the JIB, which contains information common to all processes in the job, notably pooled quotas.

Although every kernel thread in a process shares the same PHD and JIB, the addresses of these structures are stored in the KTB rather than the PCB for performance reasons.

KTB$L_QUANT contains the amount of quantum, measured in soft ticks, remaining to the kernel thread (see Section 2.6.2).

KTB$L_CPU_ID contains the processor ID of the CPU on which the kernel thread is currently executing or has last executed.

KTB$L_WAITIME contains the system absolute time in units of 10-millisecond intervals, or soft ticks, at which a kernel thread was most recently placed into a wait state.

KTB$L_ONQTIME records the system absolute time in soft ticks at which a kernel thread most recently reached quantum end.

KTB$L_MTXCNT contains the number of mutexes the kernel thread currently has locked.

The other KTB fields shown in Figure 2.3 are discussed in Section 2.5.

### 2.1.3 Scheduling State Queues

KTBs of kernel threads in most scheduling states are linked with those of other kernel threads in the same states. There are queues for computable kernel threads and for kernel threads in different wait states. The listheads for all these queues are defined in the module SYSTEM_DATA_CELLS. There is no queue of current kernel threads; instead, the per-CPU database of each CPU has a pointer to the KTB of its current kernel thread.

There are 64 queues for COM kernel threads, one for each possible priority. The quadword listheads of these queues are defined as an array whose starting address is global location SCH$AQ_COMH. A kernel thread is inserted into the queue corresponding to the internal value of its current priority (see Section 2.3). There is a similar array of 64 quadword listheads for the COMO state at global location SCH$AQ_COMOH.

The condition (empty or not) of each computable queue is summarized by a bit. If the queue contains one or more KTBs, the bit is set; if the queue is empty, the bit is clear. The 64 bits describing the COM queues are in the quadword at global location SCH$GQ_COMQS; the COMO queues are summarized in the quadword SCH$GQ_COMOQS. Bit 0 in each quadword corresponds to the queue for priority 63, bit 1 for priority 62, and so forth. (Section 2.3 explains the inverted order.) These summary quadwords facilitate selection of the next kernel thread to execute and selection of the next process to be inswapped (based on the priority of its computable kernel threads). Figure 2.4 shows the computable queues and their summary quadwords.

Figure 2.5 shows the array of scheduling wait queue headers. Each header contains a listhead for kernel threads in one of the wait states. The first two longwords are the links to the KTBs in this queue. The field WQH$L_WQSTATE contains the numeric value corresponding to the scheduling state

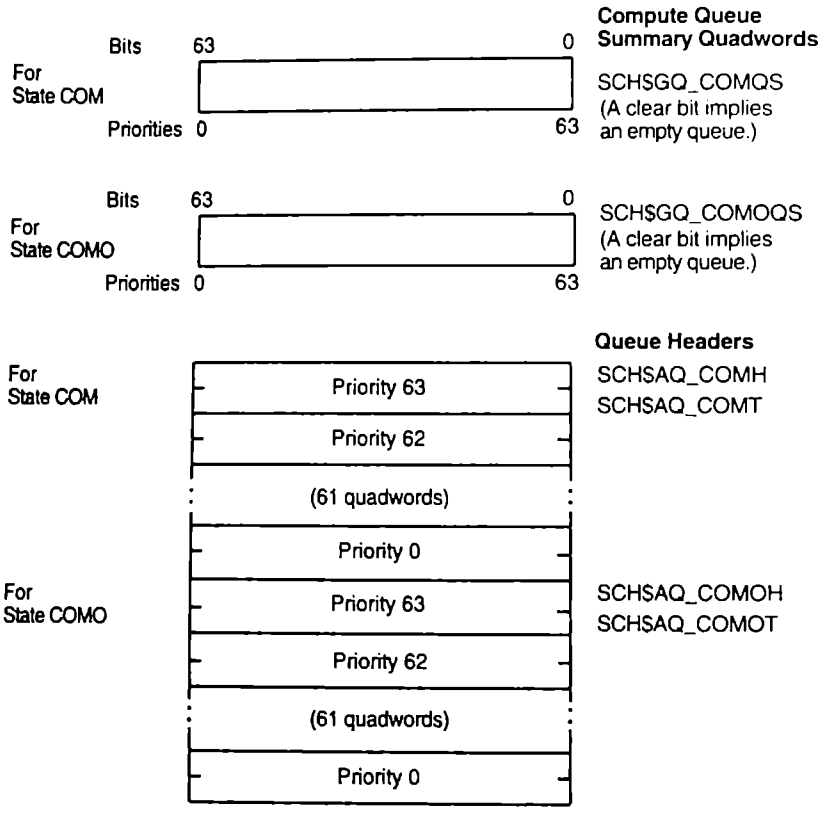



**Figure 2.4**
Computable (Executable) State Queues

of this queue (see Table 2.1 in Section 2.4). All KTBs in a state queue have KTB$L_STATE values identical to the state value of the wait queue header. The field WQH$L_WQCNT contains the number of KTBs currently in this state and queue.

The wait queue headers for all wait states except common event flag (CEF) wait are defined within this array ordered by increasing state number, with the collided page wait state first. Each wait queue header except CEF has its own global pointer. A scheduling routine can access a particular wait queue by specifying its global name or using its state number as an index into the wait queue header array. The global location SCH$AQ_WQHDR is the address of the beginning of the array and corresponds to index number 1. (There is no state whose numeric value is 0.) Note that there is no actual header with an index value of 3, or CEF, although space is reserved.

A kernel thread waiting for one or more common event flags is queued to a wait queue in the common event block (CEB) defining the common event flag cluster with which the process is associated. A CEB includes four longwords that correspond to a wait queue header. The entire format of the CEB is shown in Chapter 3. Having a wait queue in each CEB makes it easier to determine which CEF kernel threads are computable when a common

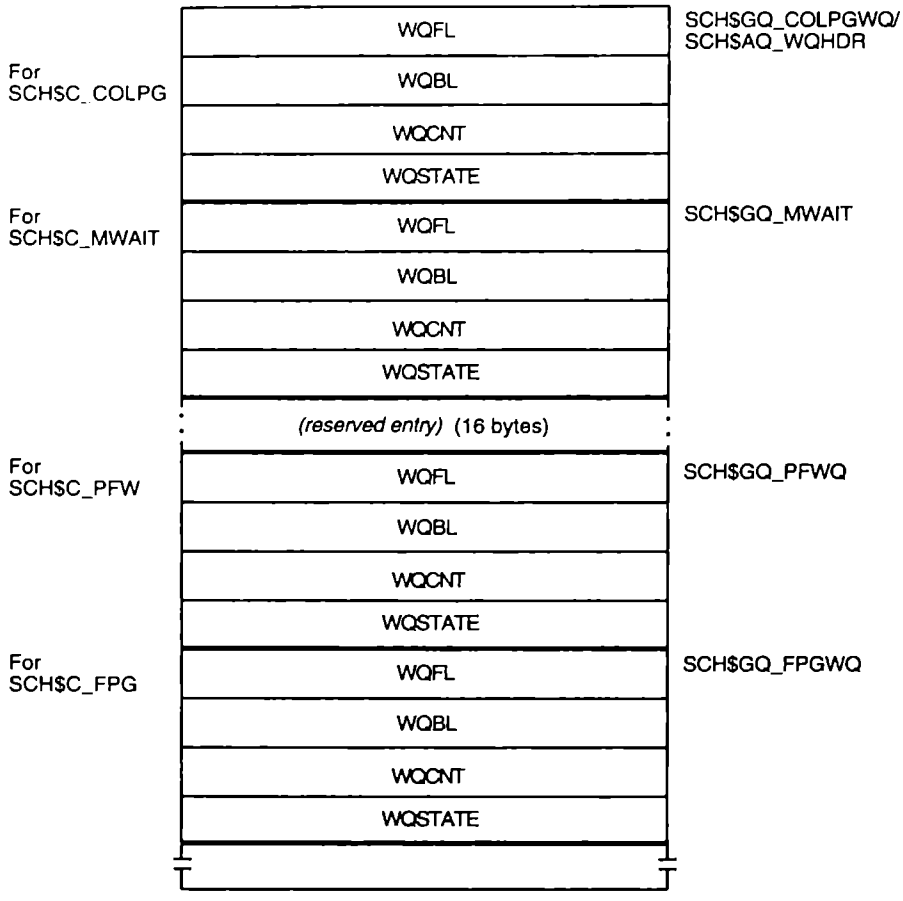



**Figure 2.5**
Array of Wait Queue Headers

event flag is set. The wait queue in the CEB contains both resident and outswapped kernel threads.

### 2.1.4 PHD and FRED Fields Related to Scheduling

Figure 2.6 illustrates some PHD and FRED fields that are particularly important for scheduling (the layout of the HWPCB is shown in Figure 2.8). Other PHD and FRED fields are shown in other chapters, in particular, Chapter *Memory Management Data Structures*.

PHD$L_CPULIM contains zero if the process is allowed unlimited CPU time, or the number of soft ticks it is allowed. PHD$L_CPUTIM contains the number of soft ticks the process has accumulated.

Each kernel thread in the process has its own FRED substructure in the PHD. Fields in the initial kernel thread's FRED can be accessed using both PHD and FRED symbolic names. Fields in other kernel threads' FREDs can only be accessed with FRED symbolic names.

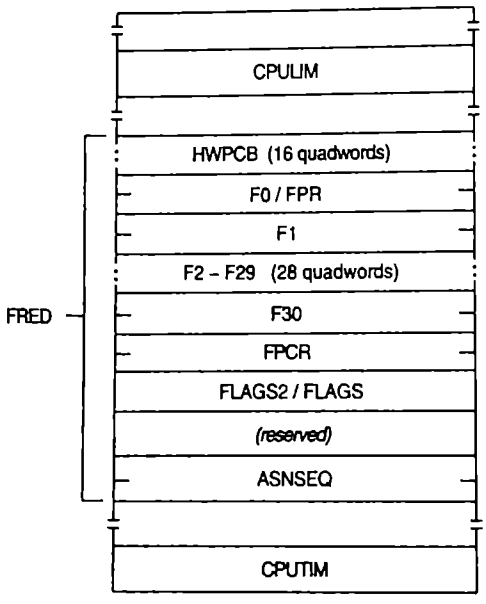



**Figure 2.6**
Process Header (PHD) Fields and Floating-Point Register and Execution Data Structure (FRED) Fields Used in Scheduling

If the image running in the process performs floating-point arithmetic, when the kernel thread is taken out of execution the contents of its floating-point registers are saved in its FRED, beginning at FRED$Q_F0. The contents of the floating-point control processor register are saved in FRED$Q_FPCR. When the kernel thread is placed back into execution, the contents of these registers are restored from the FRED.

FRED$L_FLAGS (which, for the initial kernel thread, is also accessed as PHD$L_FLAGS2) contains two flags related to scheduling:

- FRED$V_SW_FEN mirrors the state of the process's floating-point enable register. When set, it indicates that the process is performing floating-point arithmetic. Examining the FRED copy of a current process's floating-point enable register takes less time than examining the register through the `CALL_PAL MFPR` instruction.
- FRED$V_AST_PENDING, when set, means that a deliverable AST is pending for the process. Its use in optimizing AST delivery to a kernel thread being placed back into execution is described in Section 2.7.1.

FRED$Q_ASNSEQ and FRED$Q_ASN are described in Section 2.2.3.

### 2.1.5 Per-CPU Database Fields Related to Scheduling

The per-CPU database records processor-specific information such as the address of the KTB of the kernel thread current on that processor, the address of

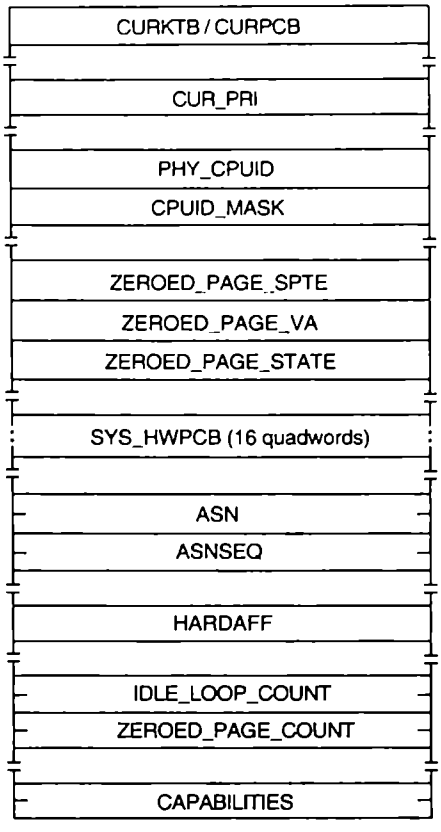



**Figure 2.7**
Per-CPU Database Fields Used in Scheduling

the processor's system context stack, and the processor's fork queues. Chapter *Symmetric Multiprocessing* contains further information, including a detailed description of the per-CPU database. Figure 2.7 illustrates the fields of the per-CPU database that are related to scheduling.

CPU$L_CURKTB (also known as CPU$L_CURPCB) contains the KTB address of the kernel thread currently executing on this processor. CPU$L_CUR_PRI contains the kernel thread's current priority. If the processor is idle, CPU$L_CURKTB contains the address of the null PCB, and CPU$L_CUR_PRI contains −1.

CPU$L_PHY_CPUID contains the ID of the processor, a number from 0 to 31. CPU$L_CPUID_MASK is a mask of all zeros with one bit set corresponding to the CPU ID.

Several fields are used by the idle loop, which executes when there is no computable resident kernel thread to run. The idle loop zeros free pages of physical memory to provide a supply for allocation of demand zero pages. During system initialization an S0/S1 space page table entry (PTE) is reserved for each processor to use in zeroing pages. The address of the PTE is recorded in CPU$L_ZEROED_PAGE_SPTE, and the starting system virtual

address mapped by that PTE, in CPU$L_ZEROED_PAGE_VA. While zeroing a page, the idle loop records its current position in CPU$L_ZEROED_PAGE_STATE. The idle loop increments CPU$Q_IDLE_LOOP_COUNT to count the number of times this processor has executed the idle loop and CPU$Q_ZEROED_PAGE_COUNT to count the number of pages this processor has zeroed. Section 2.7.3 describes the idle loop.

The Alpha architecture requires that there always be a current HWPCB to define a context for CPU execution. Beginning at CPU$Q_SYS_HWPCB is a HWPCB for use by the processor when there is no current kernel thread to run. During system initialization a HWPCB and kernel stack are allocated for each processor as part of the per-CPU data area. When there is no computable resident kernel thread to place into execution, the rescheduling interrupt service routine swaps to this context. Chapter *Symmetric Multiprocessing* contains further details.

CPU$L_CAPABILITIES, the low-order longword of CPU$Q_CAPABILITIES, is a bit mask with bits set to represent the capabilities of this processor. The low bit, when set, means that this CPU is the primary processor. The macro $CPBDEF defines symbolic values for the bits in this field. The unnamed high-order longword of CPU$Q_CAPABILITIES contains a bit mask of 31 zeros with a single bit set in the bit position corresponding to the CPU ID number. Both longwords are used to determine whether a CPU meets a kernel thread's capability and explicit affinity requirements.

CPU$L_HARDAFF is the number of kernel threads that have affinity for this CPU. Section 2.5 describes the meaning and use of these two fields.

CPU$Q_ASN and CPU$Q_ASNSEQ are described in Section 2.2.3.

## 2.2 HARDWARE CONTEXT

The definition of an executable entity from the viewpoint of the hardware is known as hardware context. It includes a virtual address space and the following registers:

- Integer registers: R0–R29, and stack pointer (SP) (SP is another name for R30, and R31 is always read as 0.)
- If the kernel thread is using floating-point, the floating-point registers: F0–F30 (F31 is always read as zero), floating-point control (FPCR) processor register, and floating-point enable (FEN) processor register
- Per kernel thread stack pointers for kernel, executive, supervisor, and user mode stacks
- Program counter (PC) and processor status (PS)
- Page table base (PTBR) processor register (see Chapter *Memory Management Overview*)
- ASN processor register
- AST enable (ASTEN) processor register (see Chapter *ASTs*)
- AST summary (ASTSR) processor register (see Chapter *ASTs*)

- Data alignment trap fixup (DATFX) processor register, which controls whether data alignment traps, which are fixed up by privileged architecture library (PALcode) routines, generate exceptions (see Chapter *Condition Handling*)
- Process cycle counter (see Chapter *Time Support*)
- Process unique value, a value used in support of multithreading software such as DECthreads
- Performance monitoring enable (PME) processor register

While a kernel thread is current, its hardware context is updated continuously. When a kernel thread is taken out of execution, its hardware context must be copied from registers to memory, where it is saved until the kernel thread executes again. OpenVMS Alpha scheduling routines ensure that the integer registers, PC, and PS are saved on the kernel thread's kernel stack and, if necessary, that the floating-point registers are saved in the FRED (see Section 2.2.2). The PALcode routine that implements the swap process context instruction, `CALL_PAL SWPCTX`, saves a subset of the hardware context in the HWPCB.

Hardware context is divided into privileged context, defined by the Alpha architecture as the contents of the HWPCB, and nonprivileged context, defined by the OpenVMS executive. The division was based upon considerations such as the following:

- Because the PALcode accesses the HWPCB physically, the HWPCB is constrained not to cross a page boundary. Keeping the HWPCB small therefore makes it easier to allocate HWPCBs without wasting physical memory.
- It is easier to change the OpenVMS executive than to update PALcode on existing machines. Thus, the SWPCTX PALcode routine saves the context necessary for an atomic context switch but not the context that could easily be saved by the executive or that might vary over time.

### 2.2.1 Hardware Privileged Context Block

Figure 2.8 shows the layout of the HWPCB. Symbolic offsets for its fields begin with the string FRED$.

The executive creates a HWPCB for each kernel thread in the fixed part of its PHD beginning at a naturally aligned 128-byte boundary. The kernel thread's KTB field KTB$Q_PHYPCB contains the physical address of its HWPCB. The swapper initializes this field in each kernel thread's KTB when it swaps a process into memory. While a kernel thread is current, the HWPCB base (PCBB) processor register contains the physical address of its HWPCB, and the processor is said to own the HWPCB. If software (other than PALcode routines) attempts read or write access to any of the HWPCB fields other than FRED$Q_ASTSR_ASTEN while the processor owns it, the results are unpredictable. (The rescheduling interrupt service routine, described in

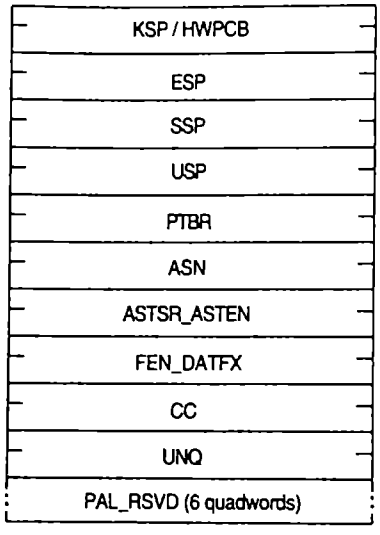



Figure 2.8
Layout of the Hardware Privileged Context Block (HWPCB)

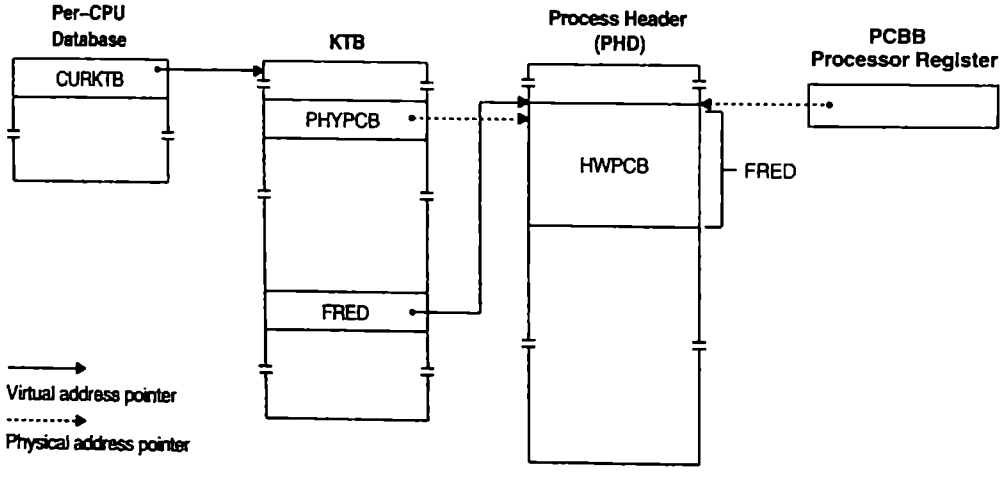



Figure 2.9
Access to the Hardware Privileged Context Block

Section 2.7.1, examines this field as a faster way to read the contents of the kernel thread's ASTSR and ASTEN registers.)

Figure 2.9 illustrates access to the HWPCB.

When the executive selects a new kernel thread to place into execution, it copies the new kernel thread's KTB$Q_PHYPCB field to R16 before executing the instruction `CALL_PAL SWPCTX`. The SWPCTX PALcode routine gets the address of the current kernel thread's HWPCB from the PCBB processor register, saves its privileged context, reloads the PCBB processor register with the address of the new kernel thread's HWPCB, and loads its privileged

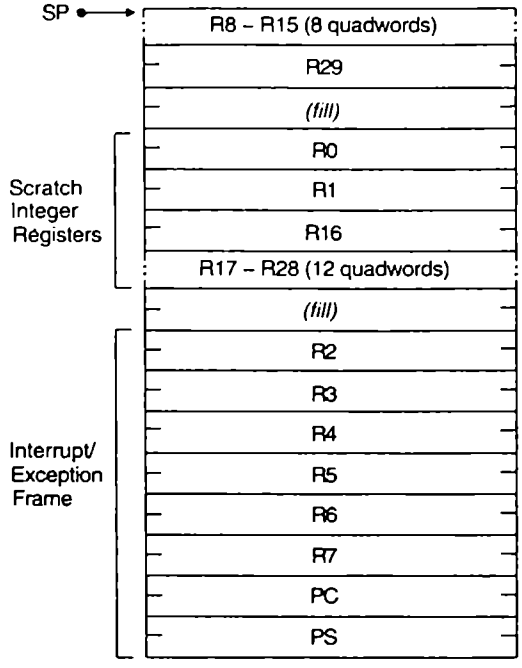



**Figure 2.10**
Arrangement of the Canonical Kernel Stack

context. The `CALL_PAL SWPCTX` instruction returns ownership of the former current kernel thread's HWPCB to the executive. Section 2.7.2 provides further details.

### 2.2.2 Canonical Kernel Stack

The term *canonical kernel stack* refers to the nonprivileged hardware context saved on a kernel stack. Figure 2.10 shows the arrangement of the canonical kernel stack.

The context is ordered based partly on the sequence that saves it in the rescheduling interrupt service routine. The oldest context (at the highest addresses) on the stack is an interrupt/exception stack frame (see Chapter *Interrupts, Exceptions, and Machine Checks*). The stack frame is followed by the scratch integer registers, saved to enable the rescheduling code to call other procedures prior to context switch without overwriting the register contents. In this chapter, the term *scratch integer registers* refers to integer registers that may be modified by a called procedure without being restored.

The quadword of fill between the interrupt/exception stack frame and scratch integer registers is present to align the stack on an octaword boundary. This alignment enables the rescheduling interrupt service routine to call other procedures after the scratch integer registers have been saved. The scratch integer registers are followed by another quadword of fill and the nonscratch integer registers other than R2 through R7. The stack alignment

enables the rescheduling interrupt service routine to call other procedures after completing the canonical kernel stack and prior to swapping this kernel thread's context.

When the kernel thread is placed back into execution by a CALL_PAL SWPCTX instruction, the rescheduling routine restores from the stack the integer registers other than R2 through R7 and then executes a CALL_PAL REI instruction to restore registers R2 through R7 and resume execution in the kernel thread (see Section 2.7.1).

For a kernel thread to be placed back into execution by the same code sequence, regardless of its state prior to the context switch, every noncurrent kernel thread must have the same arrangement of saved hardware context. Moreover, a canonical kernel stack is created for each new process's initial kernel thread as the process is created; when the rescheduling routine swaps to that kernel thread's context and executes a CALL_PAL REI instruction, it transfers control to EXE$PROCSTRT (see Chapter 7).

### 2.2.3 Address Space Numbers

Alpha address translation involves both the CPU and a PALcode routine. To optimize address translation performance, an Alpha CPU includes a cache called a translation buffer (TB). A TB caches the results of recent successful virtual address translations of valid pages. Each TB entry caches one translation: a virtual page number and, minimally, its corresponding page frame number, address space match bit, and protection bits.

The contents of a TB entry that represents a valid translation can remain valid until they are superseded by a later translation of a different virtual address that maps to the same TB entry. The OpenVMS executive is therefore responsible for flushing stale, no longer correct entries from the TB; for example, it must invalidate a TB entry corresponding to a page being deleted or removed from a process's working set.

Because all processes have the same virtual address range, all TB entries are process-specific. In theory, therefore, the entire TB would have to be invalidated whenever process context is swapped. However, a TB entry that represents a physical page shared at the same virtual address in all processes need not be invalidated. When process context is swapped, the SWPCTX PALcode routine invalidates only entries whose address space match bits are clear.

The Alpha architecture includes another feature to minimize unnecessary invalidations, called address space number (ASN). Each TB entry is tagged with a number identifying the address space whose address translation the TB entry represents. (TB entries for pages whose address space match bits are set are not tagged in this way.) The processor register ASN is part of hardware privileged context on a CPU that supports this feature. The current ASN is an implicit input for all TB lookups and invalidation of single TB entries.

On a CPU that supports ASNs, the SWPCTX PALcode routine does not invalidate TB entries. Instead, the executive ensures that unique ASNs are assigned to different kernel threads and invalidates all process-private TB entries only if an ASN in current use must be reassigned. Use of ASNs can increase the usefulness of the TB as a cache by making it possible for entries to remain cached across multiple executions of a kernel thread on a particular CPU.

ASN is a CPU-specific designation; although each member of an SMP system uses the same range of ASNs, a specific ASN on one CPU typically does not represent the same virtual address space as that ASN on another CPU. Because ASNs are CPU-specific and because multiple kernel threads of a process can execute concurrently on multiple CPUs, an ASN is associated with each HWPCB rather than with each process or address space.

The executive maintains a separate ASN counter for each CPU in its per-CPU database. CPU$Q_ASN contains the ASN most recently assigned to a kernel thread on this CPU. CPU$Q_ASN is initialized to the maximum ASN possible on the CPU. (The maximum number possible is CPU-specific; on a DEC 7000 Model 610, for example, the maximum number is $3F_{16}$, and on an AlphaServer 8400 5/300, the maximum number is $7F_{16}$.)

As the executive assigns an ASN to a kernel thread, it stores the ASN in the kernel thread's HWPCB field, FRED$Q_ASN, and decrements CPU$Q_ASN. When CPU$Q_ASN is decremented to zero and ASN 0 is assigned, all possible ASNs have been assigned on this CPU. CPU$Q_ASN is reset to its maximum possible value. At this point, the next ASN to be assigned is one that has already been assigned. If the executive were simply to assign this ASN, multiple kernel threads might be using the same ASN on the same CPU.

If multiple kernel threads were to use the same ASN concurrently, one kernel thread's virtual address could be translated using another's PTE contents, resulting in corruption of both virtual address spaces. To prevent this possibility, the executive associates a sequence number with each ASN assignment. When a kernel thread is assigned an ASN, the current sequence number, the contents of CPU$Q_ASNSEQ, are copied to its FRED$Q_ASNSEQ field.

Whenever the executive resets CPU$Q_ASN to its maximum, it increments CPU$Q_ASNSEQ as a signal that all ASNs previously assigned on this CPU are invalid and cannot be reused. It invalidates all process-private TB entries.

When the only kernel thread of a single-threaded process is taken out of execution, the ID of the CPU it ran on is stored in PCB$L_PRVCPU, and its ASN and ASNSEQ are stored in PCB$Q_PRVASN and PCB$Q_PRVASNSEQ. Whenever a single-threaded process is about to be placed into execution, the executive examines PCB$L_PRVCPU to see if the address space will be active on the same CPU as the last time this process's kernel thread executed. If so

and if the saved sequence number matches the current contents of CPU$Q_ASNSEQ, the kernel thread continues to use the same ASN. It can thus take advantage of any of its address translations still cached from the last time it ran. If it is not running on the same CPU, the kernel thread is assigned a new ASN. If the process virtual address space will be active on the same CPU, but the saved sequence number does not match the current contents of CPU$Q_ASNSEQ, the process is assigned a new ASN.

When a page of a current process is made invalid, the corresponding TB entry is invalidated. No such action occurs when a page of a noncurrent process is made invalid, for example, during swapper trimming of the working set. When invalidating pages of a noncurrent process, the executive therefore stores the reserved CPU ID value, which is −1, in the process's PCB$L_PRVCPU. This forces assignment of a new ASN the next time the process's kernel thread is placed into execution. If this step were not taken and the kernel thread ran on a CPU that had stale cached translations, it could access physical pages that had been reallocated to another use, causing memory corruption.

In a multithreaded process, when a virtual page is made invalid, each CPU on which a thread of that process is current (the CPUs represented in PCB$L_ACTIVE_CPUS) invalidates the corresponding TB entry. This prevents the kernel threads from making virtual accesses to a physical page that could have been reallocated to another use. For performance reasons, however, the invalidation is not broadcast to CPUs on which no threads of the process are current. As a result, it is possible that stale entries could remain cached on a CPU on which a thread of the process ran but is no longer current.

To prevent inadvertent reuse of such stale entries, the executive limits the circumstances in which a kernel thread can reuse an ASN from that process to the following:

- When the context of one kernel thread of a process is switched for that of another kernel thread from the same process, the newly scheduled kernel thread assumes the ASN and ASNSEQ of the one taken out of execution, thus taking advantage of cached TB entries.
- When the last current kernel thread of a multithreaded process is taken out of execution, the ID of the CPU, the ASN, and the ASN sequence number are stored in the process's PCB.

  After any time period during which no kernel thread of the process is active, when a kernel thread of that process is next placed into execution, it can use the information in PCB$Q_PRVASN and PCB$Q_ASNSEQ if the CPU ID is the same as the one recorded in PCB$L_PRVCPU and if PCB$Q_ASNSEQ matches CPU$Q_ASNSEQ.

  These checks ensure that CPU ASNs have not wrapped, resulting in reuse of ASNs, that the ASN is one that the process used on this CPU, and

that no swapper trimming of the working set occurred. The requirement that no other kernel thread of the process be active ensures that working set list replacement paging could not have resulted in invalidations that were not recorded on this CPU.

Under any other circumstances, a kernel thread placed into execution is assigned a new ASN.

## 2.3 KERNEL THREAD PRIORITY

Two different mechanisms whose names contain the term *priority* are associated with each kernel thread. Interrupt priority level (IPL) applies to kernel-thread-based and system-based code alike. IPL governs the hardware precedence of interrupts, as described in Chapter *Interrupts, Exceptions, and Machine Checks*.

Kernel thread priority determines the precedence of a kernel thread for execution and memory residence. Throughout this volume, the term *priority* used without qualification refers to kernel thread priority.

Kernel thread priorities have two different representations, an external one for presentation to the user and an internal one for use by most scheduling code. External priorities take on values from 0 to 63; 0 is the lowest priority and 63 is the highest. This representation matches the tendency of most users to associate higher values with higher priorities.

The range of 64 priorities is divided in two. The priorities from 16 to 63 are assigned to real-time kernel threads; the priorities from 0 to 15 are assigned to normal kernel threads. The scheduling of a kernel thread is affected by its type (normal or real-time) and its assigned priority level.

Traditionally, OpenVMS priorities are in the range 0 to 31, which is divided into a real-time half and a normal half. The OpenVMS Alpha priority range includes additional real-time priorities for real-time Portable Operating System Interface (POSIX) support.

Internal kernel thread priorities are stored in an inverted order. For example, 0, the lowest external priority, is stored internally as 63; external priority 63 is stored internally as 0. Subtracting one priority form from 63 converts it to the other form.

In the original operating system design, the values were inverted in this way to facilitate selection of the next process to execute and the next process to be inswapped; on OpenVMS VAX systems, these functions use the find first set (FFS) instruction, which begins its search for a set bit at bit position 0. The OpenVMS Alpha executive uses the same representation for compatibility and ease of porting. (In other data structures, external priority is used instead, for convenience of the code referencing them.) As a result of this inversion, priority promotions or boosts are implemented through subtract or decrement instructions.

System utilities, such as the System Dump Analyzer (SDA), MONITOR, and the code that implements the Digital command language (DCL) command SHOW SYSTEM, convert internal priorities to external ones for display. The Get Job Process Information ($GETJPI) system service returns an external priority when a kernel thread priority is requested.

All discussions in this volume use external priority representation unless otherwise noted. This convention should be taken into account when relating descriptions in this book to the actual routines in the listings, where internal priorities predominate.

Several fields of the KTB describe kernel thread priority. The values in these fields are in internal priority representation. The field KTB$L_PRI defines the current kernel thread priority, which is used to make scheduling decisions. KTB$L_PRIB defines the base priority of the kernel thread, from which the current priority is calculated. For normal or time-sharing kernel threads, these priority values are sometimes different, while real-time kernel threads always have identical current and base priority values.

When a process is first created, its initial kernel thread's base priority is initialized from an argument to the Create Process ($CREPRC) system service. Subsequently, if the process executes the LOGINOUT.EXE image, it may reset the base priority using the value from the user's record in the system authorization file.

A kernel thread of a process with the ALTPRI privilege can raise and lower its current and base priorities without constraint, using the Set Priority ($SETPRI) system service or the DCL command SET PROCESS/PRIORITY. Chapter 6 describes the operation of the $SETPRI system service. The field KTB$L_AUTHPRI contains the base priority authorized at the time the process was created. A kernel thread of a process without the ALTPRI privilege may raise and lower its priorities only between 0 and the contents of KTB$L_AUTHPRI.

System mechanisms that adjust priority dynamically are described in Section 2.3.3.

The fields KTB$L_PRIBSAV and KTB$L_PRISAV record the base and current priority values at the time a kernel thread first locks a mutex, before it receives a temporary elevation into the real-time range. When the kernel thread unlocks the mutex, its priority values are restored from these fields.

SCH$AQ_PREEMPT_MASK is a 64-quadword array of constants, with one quadword for each priority. The array is indexed by internal priority; the quadword at SCH$AQ_PREEMPT_MASK corresponds to internal priority 0. The quadword for a priority represents the priorities that it can preempt. Each bit in the quadword represents a priority, with bit 0 representing *external* priority 0. The bits are organized that way because they are masked against the data in SCH$GQ_ACTIVE_PRIORITY, described later in this section.

During system initialization the preemption masks that represent the rules for normal priority kernel threads are altered in accordance with the SYSGEN parameter PRIORITY_OFFSET, whose default value is 0.

When a resident kernel thread becomes computable, scheduling code must decide whether the kernel thread should preempt one currently executing. Scheduling code indexes the preemption array using the priority of the newly computable kernel thread. The selected mask indicates which priorities the kernel thread can preempt. The values in the masks implement two preemption rules and thus simplify the decision code:

- A real-time kernel thread can preempt any kernel thread of lower priority.
- A normal kernel thread at priority *n* can preempt a kernel thread at or below priority *n* – (PRIORITY_OFFSET + 1).

Altering PRIORITY_OFFSET enables the system manager to reduce the number of times a current kernel thread is preempted by a newly computable kernel thread. Preventing preemption by a newly computable kernel thread only one or two priority levels higher than a current kernel thread improves overall system performance. It minimizes unnecessary rescheduling and helps to minimize movement of a kernel thread from one processor to another on an SMP system. If no other scheduling events intervene, such a newly computable kernel thread will be favored at quantum end of the current kernel thread.

A system in which one class of users is to have significantly better responsiveness than another requires that the base priorities for the two classes must differ by the value of the SYSGEN parameter PRIORITY_OFFSET + 1.

The per-CPU database field CPU$L_CUR_PRI contains the internal form of the current priority of the kernel thread current on that CPU. If the CPU is idle and has no current kernel thread, the field contains –1.

The priorities of the kernel threads current on each member of an SMP system are described by two system data structures, defined in SYSTEM_DATA_CELLS. SCH$AL_CPU_PRIORITY is a 64-longword array with one longword for each priority. The array is indexed by internal priority; the longword at SCH$AL_CPU_PRIORITY corresponds to internal priority 0. Each bit in a longword represents one SMP member, with bit 0, for example, corresponding to CPU ID 0. Bit *m* set in longword *n* means that the kernel thread current on CPU ID *m* is at internal priority *n*.

SCH$GQ_ACTIVE_PRIORITY, summarizing SCH$AL_CPU_PRIORITY, has a bit for each priority. When set, a particular bit indicates that one or more SMP members have a current kernel thread at that priority. Bit 0, for example, corresponds to external priority 0. Originally, the quadword was designed to be indexed by external priority so that VAX scheduling code could execute a FFS instruction to locate the lowest priority current process.

Figure 2.11 shows how these data structures might look for an SMP system of two members with CPU IDs 0 and 2. CPU 0 is executing a kernel thread

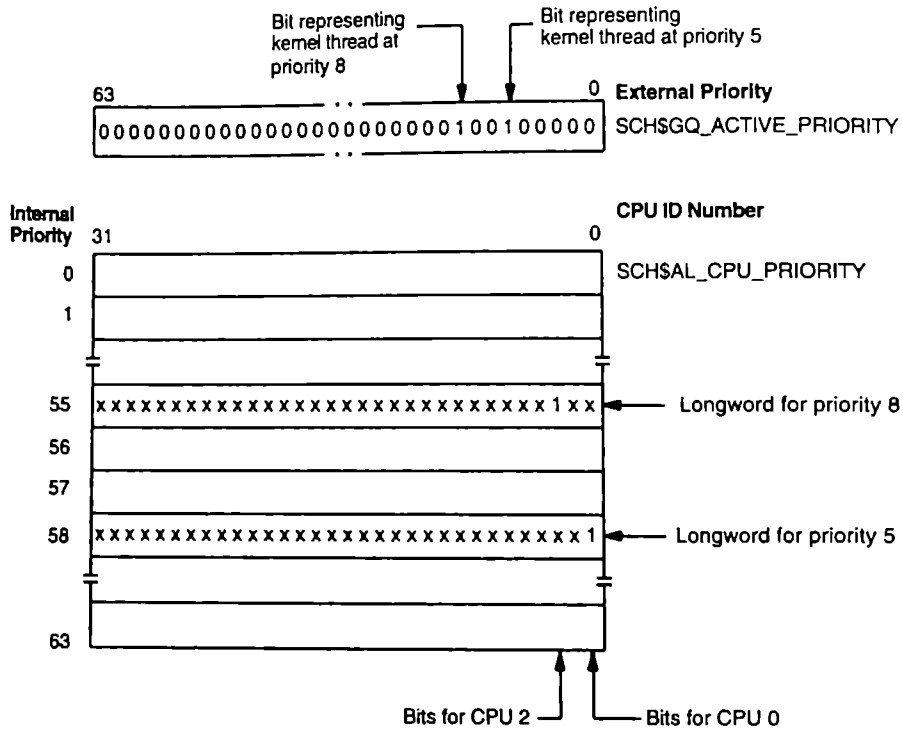



**Figure 2.11**
SMP Priority Summary Data Structures

at external priority 5, and CPU 2, a kernel thread at external priority 8. For simplicity, most of the bits are omitted.

### 2.3.1 Real-Time Priority Range

Kernel threads with priority levels 16 through 63 are considered real-time kernel threads. Two scheduling characteristics distinguish real-time kernel threads from normal kernel threads:

- The current priority of a real-time kernel thread does not change over time unless there is a direct program or operator request to change it. No dynamic priority adjustment (see Section 2.3.3) is applied by the executive.
- A real-time kernel thread executes until it is preempted by a higher priority kernel thread or it enters a wait state (see Section 2.4.3). A real-time kernel thread is not susceptible to quantum end (see Section 2.6.2); that is, it is not removed from execution because some interval of execution time has expired.

Taken in isolation, the real-time range of priorities provides a scheduling environment like traditional real-time systems: preemptive, priority-driven scheduling without a timeslice or quantum.

### 2.3.2 Normal Priority Range

Most users are authorized to have normal base priorities; the kernel threads they create thus have normal priorities. All system processes except the swapper and the Files-11 Extended QIO Processor (XQP) cache server process have kernel threads with normal priorities.

The current priority of a normal kernel thread varies over time, while its base priority remains constant unless there is a direct program or operator request to change it. This behavior is the result of dynamic priority adjustment applied by the executive to favor I/O-bound kernel threads and kernel threads performing terminal I/O over those performing other types of I/O and compute-bound kernel threads. The mechanism of priority adjustment is discussed in Section 2.3.3.

Normal kernel threads run in a time-sharing environment that allocates time slices (or quanta) to kernel threads in turn. A normal kernel thread executes until one of the following events occurs:

- It is preempted by a higher priority computable kernel thread. For one normal kernel thread to preempt another, the priority of the preempting kernel thread must be at least PRIORITY_OFFSET + 1 more than that of the preempted kernel thread.
- It enters a resource or event wait state.
- It has used its current quantum, and there is another computable kernel thread at the same or higher priority.

Kernel threads with identical current priorities are scheduled on a round-robin basis. That is, apart from the affinity and capability constraints described in Section 2.5, each kernel thread at a given priority level executes in turn before any other kernel thread at that level executes again.

Most normal kernel threads experience round-robin scheduling because, by default, the user authorization file defines the base priority for users as the value of SYSGEN parameter DEFPRI. Its usual value is 4.

### 2.3.3 Dynamic Priority Adjustment

Normal kernel threads do not generally execute at a single priority level. Rather, the priority of a normal kernel thread changes over time in a range of zero to six priority levels above the base kernel thread priority. Two mechanisms provide this priority adjustment:

- When a condition for which the kernel thread has been waiting is satisfied or a needed resource becomes available, its current priority may be increased to improve the scheduling response for the kernel thread. The size of the possible increase varies (see Section 2.6.6).
- Each time the kernel thread executes without further system events (see Section 2.6.5), the current priority is lowered by one priority level until its base priority is reached (see Section 2.7.1).

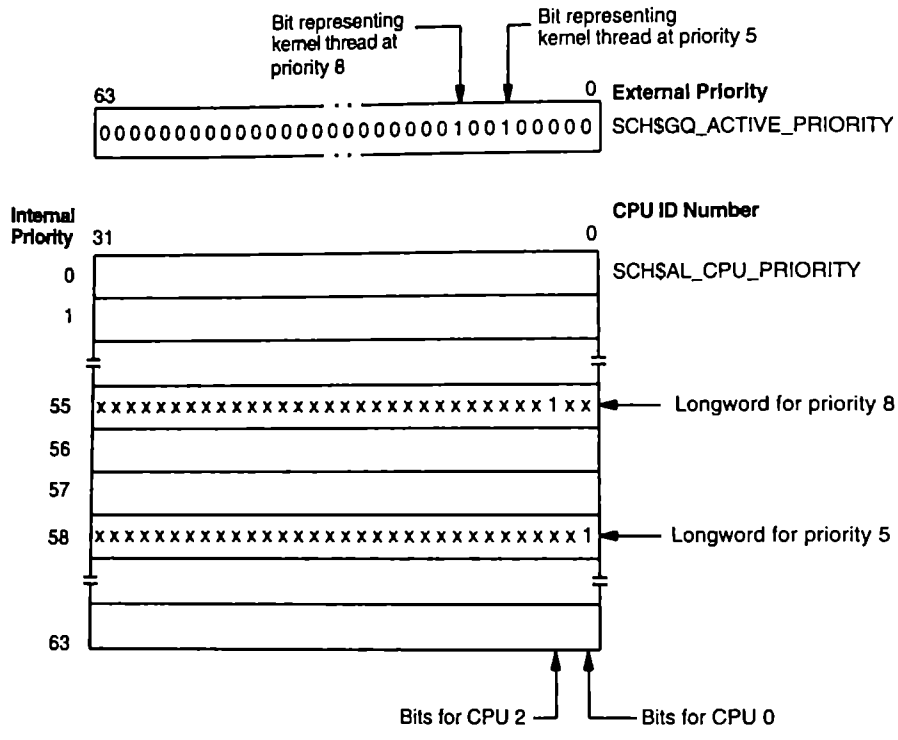



**Figure 2.11**
SMP Priority Summary Data Structures

at external priority 5, and CPU 2, a kernel thread at external priority 8. For simplicity, most of the bits are omitted.

### 2.3.1 Real-Time Priority Range

Kernel threads with priority levels 16 through 63 are considered real-time kernel threads. Two scheduling characteristics distinguish real-time kernel threads from normal kernel threads:

- The current priority of a real-time kernel thread does not change over time unless there is a direct program or operator request to change it. No dynamic priority adjustment (see Section 2.3.3) is applied by the executive.
- A real-time kernel thread executes until it is preempted by a higher priority kernel thread or it enters a wait state (see Section 2.4.3). A real-time kernel thread is not susceptible to quantum end (see Section 2.6.2); that is, it is not removed from execution because some interval of execution time has expired.

Taken in isolation, the real-time range of priorities provides a scheduling environment like traditional real-time systems: preemptive, priority-driven scheduling without a time slice or quantum.

### 2.3.2 Normal Priority Range

Most users are authorized to have normal base priorities; the kernel threads they create thus have normal priorities. All system processes except the swapper and the Files-11 Extended QIO Processor (XQP) cache server process have kernel threads with normal priorities.

The current priority of a normal kernel thread varies over time, while its base priority remains constant unless there is a direct program or operator request to change it. This behavior is the result of dynamic priority adjustment applied by the executive to favor I/O-bound kernel threads and kernel threads performing terminal I/O over those performing other types of I/O and compute-bound kernel threads. The mechanism of priority adjustment is discussed in Section 2.3.3.

Normal kernel threads run in a time-sharing environment that allocates time slices (or quanta) to kernel threads in turn. A normal kernel thread executes until one of the following events occurs:

- It is preempted by a higher priority computable kernel thread. For one normal kernel thread to preempt another, the priority of the preempting kernel thread must be at least PRIORITY_OFFSET + 1 more than that of the preempted kernel thread.
- It enters a resource or event wait state.
- It has used its current quantum, and there is another computable kernel thread at the same or higher priority.

Kernel threads with identical current priorities are scheduled on a round-robin basis. That is, apart from the affinity and capability constraints described in Section 2.5, each kernel thread at a given priority level executes in turn before any other kernel thread at that level executes again.

Most normal kernel threads experience round-robin scheduling because, by default, the user authorization file defines the base priority for users as the value of SYSGEN parameter DEFPRI. Its usual value is 4.

### 2.3.3 Dynamic Priority Adjustment

Normal kernel threads do not generally execute at a single priority level. Rather, the priority of a normal kernel thread changes over time in a range of zero to six priority levels above the base kernel thread priority. Two mechanisms provide this priority adjustment:

- When a condition for which the kernel thread has been waiting is satisfied or a needed resource becomes available, its current priority may be increased to improve the scheduling response for the kernel thread. The size of the possible increase varies (see Section 2.6.6).
- Each time the kernel thread executes without further system events (see Section 2.6.5), the current priority is lowered by one priority level until its base priority is reached (see Section 2.7.1).

Over time, compute-bound kernel thread priorities tend to remain at their base priority levels, whereas I/O-bound kernel threads tend to have average current priorities somewhat higher than their base priorities.

An example of priority adjustment that occurs over time for several kernel threads is described in Section 2.6.6 and illustrated in Figure 2.16.

A normal kernel thread occasionally has its priority boosted by the pixscan mechanism, described in Section 2.6.7.

Temporary priority adjustment can also occur as a result of locking a mutex and through action by the $GETJPI system service, which is described in Chapter 6.

## 2.4 SCHEDULING STATES

This section describes the various scheduling states and some of the transitions among them. Figure 2.12 shows the common transitions but omits a few of the less frequent ones.

The symbolic name for a scheduling state has the form SCH$C_*mnemonic*, for example, SCH$C_COM. These symbolic names are defined by the macro $STATEDEF. Table 2.1 lists the scheduling states and the corresponding KTBSL_STATE values.

Certain wait conditions are represented by two different scheduling states: one resident and one outswapped. A kernel thread waiting for a local event flag is in the LEF or the LEFO state, depending on whether its process is resident. Other scheduling states, such as CEF, include the kernel threads of both resident and outswapped processes. The KTB$V_RES bit in KTB$L_STS specifies whether the process is resident or outswapped.

### 2.4.1 Current State

A kernel thread in the CUR state is currently being executed. Its KTB address is recorded in its processor's per-CPU database at CPU$L_CURKTB.

A CUR kernel thread makes a transition to the COM state when it is preempted by a higher priority kernel thread. A CUR kernel thread of normal priority also makes this transition when it reaches quantum end and there is another computable kernel thread of higher or equal priority. A CUR kernel thread can make a transition to any of the resident wait states by directly or indirectly requesting a system operation that cannot complete immediately.

Direct requests for a system service like Hibernate ($HIBER) place the kernel thread into the voluntary wait state HIB. Direct requests for the Suspend Process ($SUSPND) system service place all the kernel threads of a process into the SUSP wait state. Direct requests for system services like Queue I/O Request and Wait ($QIOW), Synchronize ($SYNCH), and Wait for Single Event Flag ($WAITFR) place the kernel thread into the voluntary wait states LEF or CEF.

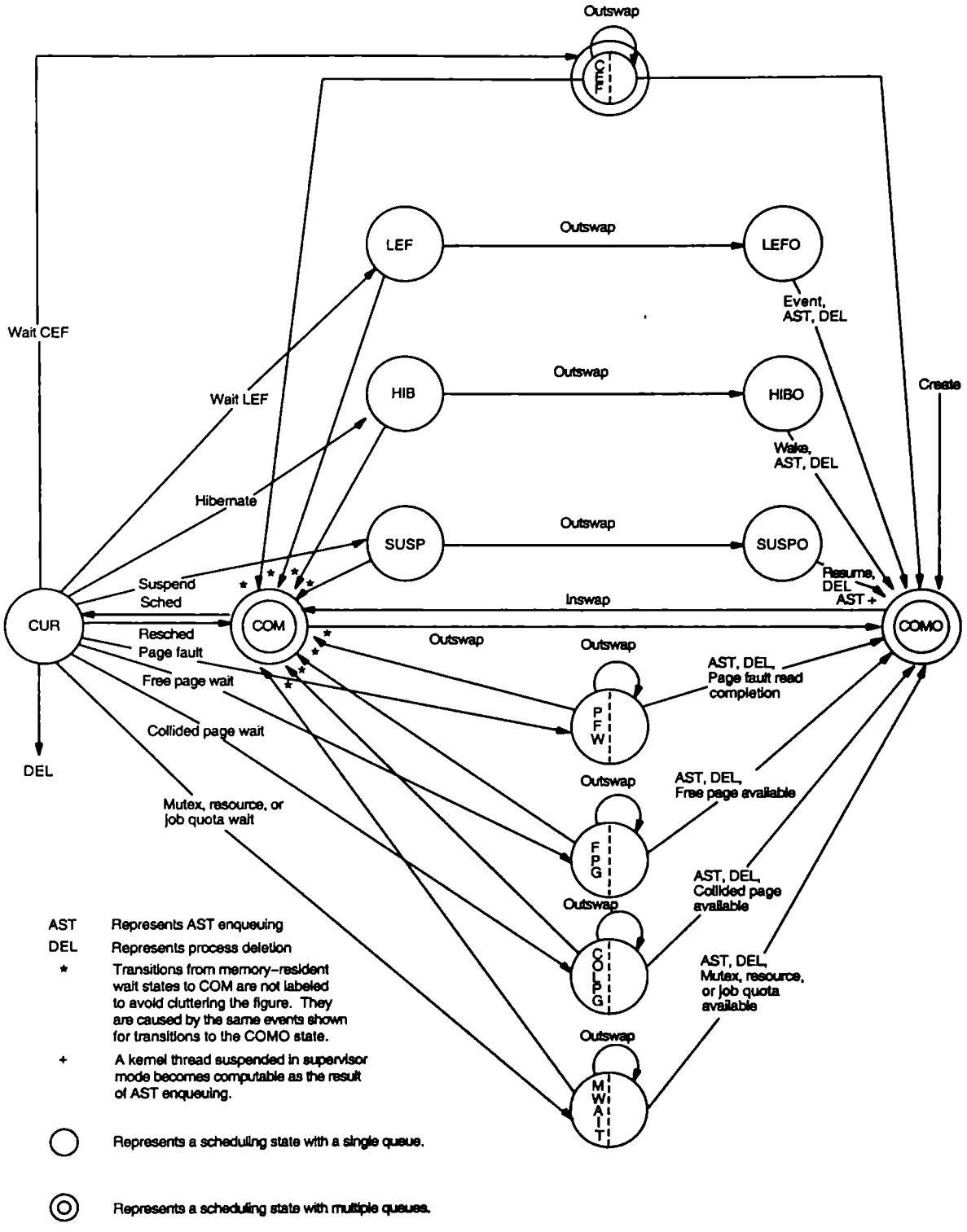



**Figure 2.12**
State Transitions

Table 2.1 Scheduling States

| *State Name* | *Mnemonic* | *Value* |
|---|---|---|
| Collided page wait | COLPG | 1 |
| Miscellaneous wait | MWAIT | 2 |
| Mutex wait | | |
| Resource wait | | |
| Job quota wait | | |
| Common event flag wait | CEF | 3 |
| Page fault wait | PFW | 4 |
| Local event flag wait (resident) | LEF | 5 |
| Local event flag wait (outswapped) | LEFO | 6 |
| Hibernate wait (resident) | HIB | 7 |
| Hibernate wait (outswapped) | HIBO | 8 |
| Suspend wait (resident) | SUSP | 9 |
| Suspend wait (outswapped) | SUSPO | 10 |
| Free page wait | FPG | 11 |
| Computable (resident) | COM | 12 |
| Computable (outswapped) | COMO | 13 |
| Currently executing process | CUR | 14 |

Indirect wait requests occur as a result of paging or contention for system resources. A kernel thread does not request PFW, FPG, COLPG, or MWAIT transitions. Rather, the transitions to these wait states occur because direct service requests to the system cannot be completed or satisfied at the moment.

Deletion of a process can only occur while its initial kernel thread is in the CUR state. The process's address space is accessible only while one of its kernel threads is current. Furthermore, process deletion in the context of the process being deleted enables the use of system services, such as Deassign I/O Channel (SDASSGN) and Delete Virtual Address Space (SDELTVA). Chapter 10 describes process deletion in detail.

### 2.4.2 Computable States

A kernel thread in the COM state is not waiting for events or resources, other than acquiring control of the CPU for execution. A COM kernel thread enters the CUR state after having been selected as the next kernel thread to run (see Section 2.7.1). A COM kernel thread enters the COMO state when its process is outswapped.

A kernel thread in the COMO state is waiting for the swapper process to bring it into memory. Then, as a COM kernel thread, it can be scheduled for execution. The initial kernel thread of a newly created process is in the COMO state.

A COM kernel thread selected for execution can enter the RWCAP miscellaneous wait state if its capability and affinity requirements have no match

on any active member of an SMP system. (This transition is not shown in Figure 2.12.) Section 2.5 describes capability and affinity requirements.

### 2.4.3 Wait States

A kernel thread that is not current or computable is waiting for the availability of a system resource or the occurrence of an event. The kernel thread is in one of several distinct wait states. The wait state reflects the particular condition that must be satisfied for the kernel thread to become computable again.

A kernel thread in a wait state makes the transition to COM or COMO through a system event such as the availability of a requested resource or the satisfaction of a wait condition. For most wait states, the queuing of an AST makes a kernel thread computable even if the wait condition is not satisfied.

**2.4.3.1 Voluntary Wait States.** Several scheduling states are associated with event flag waits: LEF, LEFO, and CEF. A kernel thread enters the LEF or CEF state as a result of requesting the $WAITFR, $SYNCH, Wait for Logical OR of Event Flags ($WFLOR), and Wait for Logical AND of Event Flags ($WFLAND) system services directly or indirectly, for example, with a $QIOW or $ENQW system service call, issued either by the kernel thread or on its behalf by some system component such as Record Management Services (RMS). A kernel thread enters the LEF state when it waits for local event flags or the CEF state when it waits for flags in a common event flag cluster. An LEF kernel thread enters the LEFO state when its process is outswapped.

The transition from the LEF, LEFO, or CEF states to the computable (COM or COMO) states can occur as a result of an event flag's being set that satisfies the wait condition, AST queuing, or process or kernel thread deletion (special cases of AST queuing). Chapter 3 describes event flag waits in more detail.

There are separate resident and outswapped states and queues for hibernating and suspended kernel threads. The $HIBER system service causes the target kernel thread to enter the HIB wait state. The $SUSPND system service causes all kernel threads of the target process to enter the SUSP state. Outswapping the process of a HIB kernel thread causes it to enter the HIBO state. Outswapping a process that has been suspended causes all its kernel threads to enter the SUSPO state.

A kernel thread makes the transition from the HIB or HIBO state to COM or COMO as a result of execution of the Wake ($WAKE) or Schedule Wakeup ($SCHDWK) system service, AST queuing, or process deletion.

The SUSP and SUSPO states are categorized by the access mode of the suspension as either supervisor mode or kernel mode. A kernel thread in supervisor mode suspension, the default, is made computable by the queuing

of an AST. (The nature of its wait, however, enables only executive and kernel mode ASTs to be delivered.) A kernel thread in kernel mode suspension is not made computable by the queuing of an AST. A kernel thread in either type of suspension is made computable when a kernel thread in another process requests the Resume Process ($RESUME) system service for the suspended process. (Suspension affects all the kernel threads in a process.) Chapter *ASTs* contains further information on ASTs, and Chapter 6, on the implementation of the $SUSPND and $HIBER system services.

Kernel thread deletion, implemented with a kernel mode AST, makes the kernel thread that is being deleted computable, even one in the SUSP or SUSPO state, because the target kernel thread is resumed before the AST is queued.

The Delete Process ($DELPRC) system service resumes a target kernel thread in case it is suspended and then queues a kernel mode AST to it. If the target kernel thread is part of a multithreaded process, the kernel mode AST requests the $END_MULTITHREAD system service, which resumes each kernel thread in the process and queues a kernel mode AST to it.

2.4.3.2 **Memory Management Wait States.** Three wait states are associated with memory management. For each there is a single queue that includes kernel threads of both resident and outswapped processes. In addition, several resource waits, described in Section 2.4.3.3.1, are associated with memory management. Memory management wait states are discussed in more detail in Chapter *Paging Dynamics*.

A kernel thread enters the page fault (PFW) wait state when code running in its context refers to a page that is not in physical memory. While the page read is in progress, the kernel thread is placed into the PFW state. Completion of the page read, AST queuing, or process deletion can cause a PFW kernel thread to become COM or COMO, depending upon whether the kernel thread's process is resident when the satisfying condition occurs. If AST queuing makes the single kernel thread of a process computable before the page read completes, then when the kernel thread reexecutes the instruction that caused the page fault, it is placed back into a page fault wait. If AST queuing makes a kernel thread in a multithreaded process computable before the page read completes, then when the kernel thread reexecutes the faulting instruction, it is placed into a collided page (COLPG) wait state.

Usually a kernel thread enters the free page (FPG) wait state when it requests a physical page to be added to its working set but there are no free pages to be allocated from the free page list. In addition, a kernel thread requesting a lock through the Enqueue Lock Request ($ENQ) system service can be placed into this wait state when the lock ID table is full and a page of physical memory to extend it cannot be allocated. This state is essentially a resource wait that ends when the supply of free pages is replenished through modified page writing, working set trimming, process outswapping, or vir-

tual address space deletion. When a physical page becomes available, all FPG kernel threads are made COM or COMO. The first kernel thread to execute allocates the page. If the free page list is empty as a result, when the other kernel threads execute, they are placed back into a FPG wait.

A kernel thread enters the COLPG wait state when more than one kernel thread causes page faults on the same shared page at the same time. The initial faulting kernel thread enters the PFW state, while the second and succeeding kernel threads enter the COLPG state. All COLPG kernel threads are made COM or COMO when the read operation completes.

2.4.3.3 **Miscellaneous Wait State.** A kernel thread in the miscellaneous wait (MWAIT) state waits for the availability of a depleted system resource or job quota or a locked mutex. The contents of the field KTB$L_EFWM identify the entity for which the process waits:

- A small positive integer generally identifies a system resource. In the case of a kernel thread waiting to acquire its process's inner mode semaphore, the low-order word of KTB$L_EFWM contains the value RSN$_INNER_MODE, and the high-order word contains the index of the process, the low-order word of its internal process ID (IPID).
- The system virtual address of the process's JIB specifies that the process is waiting for a job quota.
- The system virtual address of a mutex specifies that the process is waiting for that mutex.

There is a single MWAIT queue for the kernel threads of both resident and outswapped processes.

2.4.3.3.1 **System Resource Miscellaneous Waits.** A kernel thread may enter a resource wait if a resource it needs is not available. Common examples are the depletion of nonpaged pool or an already full mailbox. The kernel thread becomes computable when an executive routine declares the resource available. AST enqueuing makes the kernel thread computable, temporarily at least (see Section 2.6.1.5).

Table 2.2 lists the resources associated with the MWAIT state. Their symbolic values are defined by the $RSNDEF macro. System utilities such as SDA, MONITOR, and the DCL command SHOW SYSTEM display the state of a kernel thread in a resource wait using one of the mnemonic names in this table.

The system global SCH$GL_RESMASK summarizes the system resources for which kernel threads in the MWAIT state are currently waiting. For example, bit 3 corresponds to RSN$_NPDYNMEM. When set, it indicates that one or more KTBs are in the MWAIT queue waiting for nonpaged pool to become available.

Table 2.2 Types of Resource MWAIT States

| *Resource Wait Name* | *Mnemonic* | *Symbolic Name* | *Number* |
|---|---|---|---|
| AST wait (wait for system or special kernel mode AST) | RWAST | RSN$_ASTWAIT | 1 |
| Mailbox full | RWMBX | RSN$_MAILBOX | 2 |
| Nonpaged dynamic memory | RWNPG | RSN$_NPDYNMEM | 3 |
| Page file full | RWPFF | RSN$_PGFILE | 4 |
| Paged dynamic memory | RWPAG | RSN$_PGDYNMEM | 5 |
| Breakthrough [1] | RWBRK | RSN$_BRKTHRU | 6 |
| Image activation lock [1] | RWIMG | RSN$_IACLOCK | 7 |
| Job pooled quota [1] | RWQUO | RSN$_JQUOTA | 8 |
| Lock identifier [1] | RWLCK | RSN$_LOCKID | 9 |
| Swap file space [1] | RWSWP | RSN$_SWPFILE | 10 |
| Modified page list empty | RWMPE | RSN$_MPLEMPTY | 11 |
| Modified page writer busy | RWMPB | RSN$_MPWBUSY | 12 |
| Distributed lock manager wait | RWSCS | RSN$_SCS | 13 |
| VMScluster transition | RWCLU | RSN$_CLUSTRAN | 14 |
| CPU capability | RWCAP | RSN$_CPUCAP | 15 |
| VMScluster server process | RWCSV | RSN$_CLUSRV | 16 |
| Reserved [1] | RWSNP | RSN$_SNAPSHOT | 17 |
| POSIX fork creation | PSXFR | RSN$_PSXFRK | 18 |
| Inner mode semaphore | INNER_MODE | RSN$_INNER_MODE | 19 |
| Exit handler for kernel thread [1] | EXH | RSN$_EXH | 20 |

[1] This resource wait is not currently used.

When an executive routine releases a resource for which kernel threads might be waiting, it typically calls SCH$RAVAIL, in module MUTEX, to declare the resource available for any waiting kernel threads. SCH$RAVAIL tests and clears the bit in SCH$GL_RESMASK corresponding to the resource. If the bit was set, the routine scans the MWAIT queue for kernel threads whose KTB$L_EFWM field matches the available resource. For any such kernel thread it finds, it makes the kernel thread COM or COMO, as appropriate.

RWAST is a general-purpose resource wait used primarily when the wait is expected to be satisfied by the queuing or delivery of an AST to the kernel thread. There is no concrete resource corresponding to the name RSN$_ASTWAIT. The $QIO system service can place a kernel thread into this resource wait when the kernel thread is not allowed to issue another buffered or direct I/O request until one completes. Another use of RSN$_ASTWAIT is to wait for all the I/O requests on a channel to complete when the $DASSGN system service is requested. A kernel thread about to be suspended waits for the RSN$_ASTWAIT resource until all the process's Files-11 XQP activity completes (see Chapter *ASTs*).

A kernel thread is placed into RWMBX wait when it has resource wait mode enabled and tries to write to a mailbox that is full or has insufficient buffer space.

A kernel thread is placed into RWNPG wait when it is unsuccessful in allocating nonpaged pool. With the expandability of nonpaged pool, this wait is relatively rare.

A kernel thread is placed into RWPAG wait when it is unsuccessful in allocating paged pool.

A kernel thread in RWMPE wait is waiting for the modified page writer to signal that it has flushed the modified page list. The only kernel thread placed into this wait is the initial kernel thread of a process running the OPCCRASH image, which forces a flush of the modified page list prior to stopping the system.

A kernel thread is placed into RWPFF wait when it faults a modified page with page file backing store out of its working set and the associated page file has not yet been initialized.

A kernel thread that faults a modified page out of its working set can be placed into RWMPB wait when either of the following is true:

- The modified page list contains more pages than the SYSGEN parameter MPW_WAITLIMIT.
- The modified page list contains more pages than the SYSGEN parameter MPW_LOWAITLIMIT and the modified page writer is active, writing modified pages.

Generally, this resource wait occurs on a system whose modified page list has grown faster than it could be written. A kernel thread in such a wait becomes computable when enough modified pages have been written so that there are MPW_LOWAITLIMIT or fewer pages left on the list.

Chapter *Paging Dynamics* contains further details on RWPFF and RWMPB waits.

The distributed lock manager uses RWSCS to stall execution of a kernel thread on a VMScluster node when the lock manager must wait for a response from a remote system that has information about a particular lock resource. This is typically a short-lived wait state.

A kernel thread that issues any lock requests on any node of a VMScluster in transition (that is, while a node is being added or removed) is placed into RWCLU wait until the VMScluster membership stabilizes.

A computable kernel thread that requires one or more CPU capabilities that cannot all be satisfied by a single active member of the SMP system is placed into RWCAP wait (see Section 2.5).

A maximum number of outstanding transfer requests may be made from one VMScluster node to a remote node's cluster server process. When this limit has been reached and a kernel thread requests a service that would initiate another such transfer, the kernel thread is placed into RWCSV wait

until transfer requests complete. Also, when a kernel thread requests certain clusterwide process system services, it is placed into RWCSV wait until the request completes on the other node.

The resource wait PSXFR is implemented to support the layered POSIX subsystem. It provides synchronization between a parent and child process during a POSIX fork operation.

A kernel thread in a multithreaded process must acquire the process's inner mode semaphore before executing kernel mode or executive mode code. If the semaphore is already owned by another kernel thread in this process, the requesting kernel thread is placed into INNER_MODE wait until the semaphore is released.

The Set Resource Wait Mode ($SETRWM) system service can cause a subsequent system service to return an error status rather than placing the kernel thread into the MWAIT state. The $SETRWM system service sets the PCB$V_SSRWAIT bit in PCB$L_STS to disable resource waits or clears it to enable them. This service affects all kernel threads in the process.

Disabling resource waits affects many directly requested operations (such as I/O requests or timer requests) but has no effect on allocation requests by the system on behalf of the user. Although a kernel thread can respond to a depleted resource error from a system service call or an RMS request, it has no means of reacting to a similar error in case of an unexpected event such as a page fault. For example, when the page fault service routine is unable to allocate an I/O request packet for a page read, it places the kernel thread into an MWAIT wait regardless of the value of PCB$V_SSRWAIT.

2.4.3.3.2 **Mutex Miscellaneous Waits.** A system routine that accesses data structures protected by a mutex places a kernel thread into the MWAIT state if the requested mutex ownership cannot be granted. Thus, the mutex wait state indicates a locked resource and not necessarily a depleted one. When the mutex is unlocked, each kernel thread waiting to lock that mutex is made COM or COMO to repeat its attempt to lock the mutex. AST queuing makes a mutex-waiting kernel thread computable only temporarily; the IPL at which the kernel thread is waited is 2, blocking the AST delivery interrupt. The kernel thread repeats its attempt to lock the mutex, and if it is still locked, the kernel thread is placed back into a wait.

Chapter *Synchronization Techniques* lists the names of mutexes whose addresses can be stored in KTB$L_EFWM and describes the mutex lock and unlock routines. System utilities such as SDA, MONITOR, and the DCL command SHOW SYSTEM display the state of a kernel thread waiting for a mutex as MUTEX.

2.4.3.3.3 **Job Quota Miscellaneous Waits.** Another type of miscellaneous wait is a wait for a depleted job quota. Currently, there are two job quotas for which a kernel thread may have to wait:

- Buffered I/O byte count quota—used in a large number of ways, including I/O requests buffered in nonpaged pool, temporary mailboxes, and window control blocks
- Timer queue entry (TQE) quota—used for timer requests and common event flag cluster creation

When a job has one or more kernel threads in such a wait, the field JIB$L_FLAGS has a bit set to indicate each job quota for which kernel threads in that job are waiting. Bit 0, when set, means that one or more kernel threads are waiting for JIB$L_BYTCNT. Bit 1, when set, means that one or more kernel threads are waiting for TQE quota.

When another kernel thread in the job returns one of these quotas, the corresponding bit is checked to see if there is any waiting kernel thread. If there is, the waiting kernel thread is made computable to repeat its attempt to charge against the job quota.

A kernel thread in a job quota wait has the address of its JIB in KTB$L_EFWM. System utilities such as SDA, MONITOR, and the DCL command SHOW SYSTEM display the state of a kernel thread waiting for a jobwide resource as MUTEX.

## 2.5 CAPABILITIES AND AFFINITY

### 2.5.1 System-Defined and User-Defined Capabilities

A capability represents a CPU attribute that a given kernel thread requires in order to execute, for example, a necessary hardware feature. In an SMP system, a kernel thread's requirement for a particular capability may limit its execution to a subset of the available processors. For example, a kernel thread might require the capability to execute on the primary processor.

OpenVMS Alpha defines three system capabilities:

- CPB$V_RUN, when set, means the CPU can schedule kernel threads. The run capability is removed from a processor by the image that implements the DCL command STOP/CPU.
- CPB$V_QUORUM, when set, means that the CPU (or the entire SMP system) is an active member of a VMScluster that has quorum. The VMScluster connection manager removes the quorum capability from every cluster member when quorum has been lost and restores it when quorum is regained. Because most kernel threads are created requiring this capability, removing this capability from a CPU blocks the execution of most kernel threads on that CPU. This mechanism makes it possible for kernel threads to be created that do not require the capability and that can thus take part in the cluster transition. No use is currently made of this mechanism.
- CPB$V_PRIMARY, when set, means that the CPU is the primary member of an SMP system or the only CPU in a uniprocessor configuration.

OpenVMS Alpha Version 7.0 adds support for additional user-defined capabilities. The Modify CPU User Capabilities (SCPU_CAPABILITIES) system service enables a privileged user to add or remove user-defined capabilities to a particular CPU, all active CPUs, or the default mask. The meaning of a particular user-defined capability is application-defined.

Each processor's per-CPU database field CPU$L_CAPABILITIES describes its set of capabilities. Each set bit in CPU$L_CAPABILITIES represents a capability of that CPU. System-defined capabilities are in the low-order word, and user-defined capabilities are in the high-order word. When a new CPU joins the SMP system, its capability mask is copied from the system default one, SCH$GL_DEFAULT_CPU_CAP. The default CPU capabilities are CPB$V_RUN and CPB$V_QUORUM. The capability of being primary is set at system initialization in the primary processor's capability mask.

The 32-longword array SCH$AL_CPU_CAP, indexed by CPU ID, collects the capability mask for all CPUs, simplifying a search for a CPU with a set of particular capabilities.

The routines SCH$ADD_CPU_CAP and SCH$REMOVE_CPU_CAP, both in module CAPABILITY, provide for dynamic changes to CPU capabilities. The contents of SCH$GL_CAPABILITY_SEQUENCE indicate to which generation the data in SCH$AL_CPU_CAP belongs; whenever the data change by the addition or removal of a CPU capability, SCH$GL_CAPABILITY_SEQUENCE is incremented. Keeping track of the generation enables the executive to detect changes in the set of processors available to meet the capability requirements of kernel threads.

Fields in each kernel thread's KTB describe its current and permanent capability requirements. KTB$L_CAPABILITIES and KTB$L_PERMANENT_CAPABILITIES record the current and permanent capability requirements. When a process is created, the permanent capability mask of its initial kernel thread is copied from the system default one, SCH$GL_DEFAULT_PROCESS_CAP, which is currently defined so as to require that the kernel thread execute on a CPU with quorum and the run capability. When a multithreaded process creates another kernel thread, the new kernel thread's current and permanent capabilities are copied from those of the creator thread.

OpenVMS Alpha Version 7.0 provides the Modify Process User Capabilities ($PROCESS_CAPABILITIES) system service to enable a privileged user to modify either the required capabilities of a particular kernel thread or the system default capabilities. This service, in combination with $CPU_CAPABILITIES, provides a supported way for an application to be bound to a particular subset of the CPUs in an SMP system.

The routines SCH$SET_CAPABILITY and SCH$CLEAR_CAPABILITY, both in module CAPABILITY, provide for dynamic changes to kernel thread capability requirements. These routines are called by the $PROCESS_CAPABILITIES system service.

The field KTB$L_CURRENT_AFFINITY is calculated whenever the kernel thread changes its capability or affinity requirements, whenever the capabilities of the current SMP members change, and whenever the SMP membership changes. It is a mask with bits set to represent the CPUs for which the kernel thread has explicit affinity that also have the capabilities the kernel thread requires. These are the CPUs on which the kernel thread can execute.

Whenever KTB$L_CURRENT_AFFINITY is calculated, the current value of SCH$GL_CAPABILITY_SEQUENCE is copied to KTB$L_CAPABILITIES_SEQ for future use as a validity check on KTB$L_CURRENT_AFFINITY.

Fields KTB$L_CAPABILITY and CPU$L_CAPABILITY are present for temporary upward compatibility. They mirror KTB$L_CAPABILITIES and CPU$L_CAPABILITIES so that existing code can read the information using these earlier field names, but they may be removed in a future release.

### 2.5.2 Explicit Affinity

Affinity is the requirement that a kernel thread execute on a subset of the CPUs in an SMP system. OpenVMS provides for both explicit and implicit affinities. A kernel thread must explicitly request explicit affinity and must explicitly dismiss it. Explicit affinity might allow kernel threads to be segregated by CPU.

Fields in each kernel thread's KTB describe its explicit current and permanent affinities. Each is a bit mask with a bit set to represent each CPU for which the kernel thread has explicit affinity. KTB$L_AFFINITIES and KTB$L_PERMANENT_AFFINITIES contain the current and permanent affinities.

OpenVMS Alpha Version 7.0 provides the Modify Process Affinity ($PROCESS_AFFINITY) system service to enable a privileged user to specify an explicit affinity mask for a particular kernel thread.

SCH$SET_AFFINITY, the executive routine that underlies the $PROCESS_AFFINITY and Diagnose ($DIAGNOSE) system services, can be called to request that a particular kernel thread acquire current or permanent explicit affinity for a particular set of CPUs. The procedure stores a new value in the affected kernel thread's KTB$L_AFFINITIES and calculates a new value for KTB$L_CURRENT_AFFINITY. Each specified processor's CPU$L_HARDAFF is incremented as a count of kernel threads that have explicit affinity for it. If the request was for permanent affinity, the procedure also stores the CPU bit mask in KTB$L_PERMANENT_AFFINITIES.

### 2.5.3 Executive Changes to Capabilities and Explicit Affinity

Following are examples of executive routines that employ capabilities and explicit affinity:

- The Get Environment Variable ($GETENV) system service must run on the primary processor when it calls a console subsystem procedure to get

the current value of a particular environment variable (see Chapter *Bootstrap Processing*).

- The undocumented $DIAGNOSE system service enables a kernel thread with DIAGNOSE privilege to acquire or remove explicit affinity to a specified CPU or to whichever CPU is primary. Use of this service is reserved to Digital; any other use is unsupported.
- The Set Time ($SETIME) system service must run on the primary processor when it reads and writes the battery-backed watch.
- The routine IOCSLAST_CHAN, in module IOSUBNPAG, entered when the last channel is deassigned to a device, tests whether the device's registers can only be accessed from a limited set of processors. If so, the routine acquires explicit affinity for the primary processor, which is always presumed to be a member of every device's affinity set. Chapter *Symmetric Multiprocessing* describes the concept of device affinity set in more detail.
- The interval timer interrupt service routine runs on each SMP member but performs system timekeeping functions only on the primary processor. It tests the low bit of the current processor's per-CPU database field CPU$L_CAPABILITIES to determine whether it is running on the primary.

At image rundown, the capability mask of a process's initial kernel thread is restored from its permanent capability mask and its affinity from the permanent affinity mask. Explicit affinity counts in the per-CPU database are adjusted. If there is any change in required capabilities or affinity, the image rundown routine requests an IPL 3 interrupt for the scheduling code to determine where the kernel thread should run.

At deletion of a kernel thread with explicit affinity, the CPU$L_HARDAFF field of any processor for which the kernel thread had explicit affinity is decremented.

During certain limited sets of circumstances, the executive saves a kernel thread's current required capabilities and affinities in KTB$L_SAVED_CAPABILITIES and KTB$L_SAVED_AFFINITIES and reinitializes KTB$L_CAPABILITIES from the system capabilities in SCH$DEFAULT_PROC_CAP plus additional specified capabilities. It clears KTB$L_AFFINITIES. It also sets bit KTB$V_SCHED_CONTEXT_SAVED in KTB$L_FLAGS to indicate that the kernel thread's CPU scheduling information has been saved. While the bit is set, any attempt to alter the kernel thread's CPU scheduling information through the previously described system services changes the saved rather than the current information. The basic idea is to prevent temporary executive capability requirements from permanently modifying user-specified requirements. One such circumstance is the execution of the SSETIME system service, which must execute on the primary processor.

### 2.5.4 Implicit Affinity

A kernel thread may have implicit affinity for a processor when there are advantages to its continuing execution on that processor. For example, a kernel thread that executed on a CPU with a large physical memory cache might have data still cached if it were placed back into execution on that CPU. Or a kernel thread placed back into execution on the same CPU on which it last executed might have process-private TB entries still cached.

OpenVMS Alpha Version 7.0 provides the Modify Process Implicit Affinity ($SET_IMPLICIT_AFFINITY) system service to enable a privileged user to request that implicit affinity be enabled or disabled for a particular kernel thread.

By default implicit affinity is disabled. When a user enables it for a particular kernel thread, the executive has more flexibility in scheduling the kernel thread to run on a CPU in which the thread's memory and address translations are more likely to be cached.

The $SET_IMPLICIT_AFFINITY system service calls procedure SCH$ACQUIRE_AFFINITY, in module CAPABILITY, to request that a target kernel thread acquire implicit affinity for a particular CPU. The routine stores the specified CPU ID in KTB$L_BIAS_CELL, whose default value is −1. It sets CPB$V_IMPLICIT_AFFINITY in the target kernel thread's KTB$L_CAPABILITIES to indicate that the kernel thread has enabled implicit affinity for the CPU in KTB$L_BIAS_CELL.

When a kernel thread with implicit affinity is selected for execution, if its affinity is not for the current CPU, the scheduling code returns the kernel thread to the compute queue and attempts to select a kernel thread to run whose priority is high enough so that it could not be preempted by the kernel thread with implicit affinity.

Potentially, the kernel thread can be skipped for execution in this manner repeatedly up to the number in KTB$L_AFFINITY_SKIP, which is decremented at each failed attempt. KTB$L_AFFINITY_SKIP is initialized from the SYSGEN parameter AFFINITY_SKIP, whose default value is 2. When KTB$L_AFFINITY_SKIP reaches 0 or when an alternative kernel thread cannot be found that can execute on this CPU, implicit affinity is broken. That is, a kernel thread's having implicit affinity for one CPU is not a compelling reason to leave another CPU idle.

## 2.6 SCHEDULING DYNAMICS

In general, on an OpenVMS system in equilibrium, the available processors execute the highest priority COM kernel threads. A number of events can alter this equilibrium and trigger rescheduling, that is, selecting another kernel thread to run and switching its context with that of the current kernel

thread, taking the current kernel thread out of execution and placing the new one into execution.

The following are the principal events that require rescheduling:

- A current kernel thread goes into a wait state.
- A current kernel thread reaches the end of its quantum, and there is another COM kernel thread of equal or higher priority.
- A current kernel thread requests explicit rescheduling through the Initiate Rescheduling Interrupt (SRESCHED) system service.
- A current kernel thread changes its priority, and there is a higher priority COM kernel thread.
- There is no longer a match between the capabilities required by a current kernel thread and the processor on which it is executing.
- A current kernel thread requests explicit affinity for a set of CPUs that does not include the current processor.
- A system event alters the scheduling state of a noncurrent kernel thread to COM, and its priority permits it to preempt a current kernel thread.

Figures 2.14, 2.15, and 2.17 show the relations among the routines involved in these events. The sections that follow describe the events and the routines that handle them. Section 2.7 describes the rescheduling interrupt.

### 2.6.1 Placing a Current Kernel Thread into a Wait State

When a kernel thread directly or indirectly requests a system operation for which it must wait, the kernel thread is placed into a wait state. Placing a kernel thread into a wait state consists of making the associated scheduler database changes and saving the kernel thread's hardware context. Saving the kernel thread's hardware context requires saving integer registers on its kernel stack to complete the canonical kernel stack (see Section 2.2.2) and possibly saving floating-point registers before switching kernel thread context. Although only MACRO-64 routines can arrange the stack precisely in this way, MACRO-32 routines can update the scheduler database.

Several routines update the scheduler database so as to place a kernel thread into a particular wait state; several form the canonical kernel stack and transfer control to the scheduling code that performs the actual context switch; and several perform all three functions. Which of these is used depends upon

- The mode in which the kernel thread is to wait
- The wait state into which the kernel thread is to be placed

To illustrate the necessary steps, this section describes one particular sequence, which waits a kernel thread in an outer mode.

When a kernel thread requests a system service that must place the kernel thread into a wait for some event or resource before it can complete the

request, the service procedure typically waits the kernel thread in the mode from which the service was requested.

If a user mode thread manager has been registered in the process, the OpenVMS Alpha Version 7.0 executive notifies it when a kernel thread is about to be waited in user mode because a user mode thread running in the context of that kernel thread is stalled. The user mode thread manager may switch the stalled user mode thread for one that is runnable, and as a result, the kernel thread is not waited. When an event occurs that would unblock such a stalled user mode thread, the executive notifies the thread manager, which can then switch that user mode thread to execute again in the context of any available kernel thread. Chapter 5 discusses these notifications, known as upcalls, and the interactions between the executive and a user mode thread manager.

Waiting a kernel thread in an outer mode enables it to receive ASTs for that mode and more privileged ones. In order to wait the kernel thread in the outer mode, the service procedure must return the status SS$_WAIT_CALLERS_MODE to its caller, the change mode dispatcher, requesting that it clean up the stack and wait the kernel thread. Each system service that can return this status must have been described as such when it was loaded, and the description must further specify whether the service is to be reexecuted when it is placed back into execution.

Figure 2.13 shows some of the main steps in a typical sequence leading to this type of kernel thread wait. In this example, no user mode thread manager is registered in the process. The numbers in the figure correspond to those in the following list:

① The kernel thread requests the system service by calling its system service transfer routine.
② The transfer routine executes a `CALL_PAL CHMK` instruction, generating an exception. The exception service routine for CHMK exceptions is the change mode dispatcher, EXE$CMODKRNL in module SYSTEM_SERVICE_DISPATCHER. The exception frame created by this exception forms the first part of a potential canonical kernel stack.
③ Determining that the service is one that may return SS$_WAIT_CALLERS_MODE and one that does not itself restore the argument registers, the change mode dispatcher saves the service arguments on the kernel stack before calling the service procedure.
④ When the service procedure determines that the kernel thread must wait, it first acquires the SCHED spinlock and then typically calls SCH$WAIT_PROC, in module SCHED_ROUTINES, to update the scheduler database.
⑤ SCH$WAIT_PROC is entered in kernel thread context and with the SCHED spinlock held. Its arguments specify the address of the KTB of the current kernel thread and the address of the wait queue into which the KTB is to be inserted. It takes the following steps:

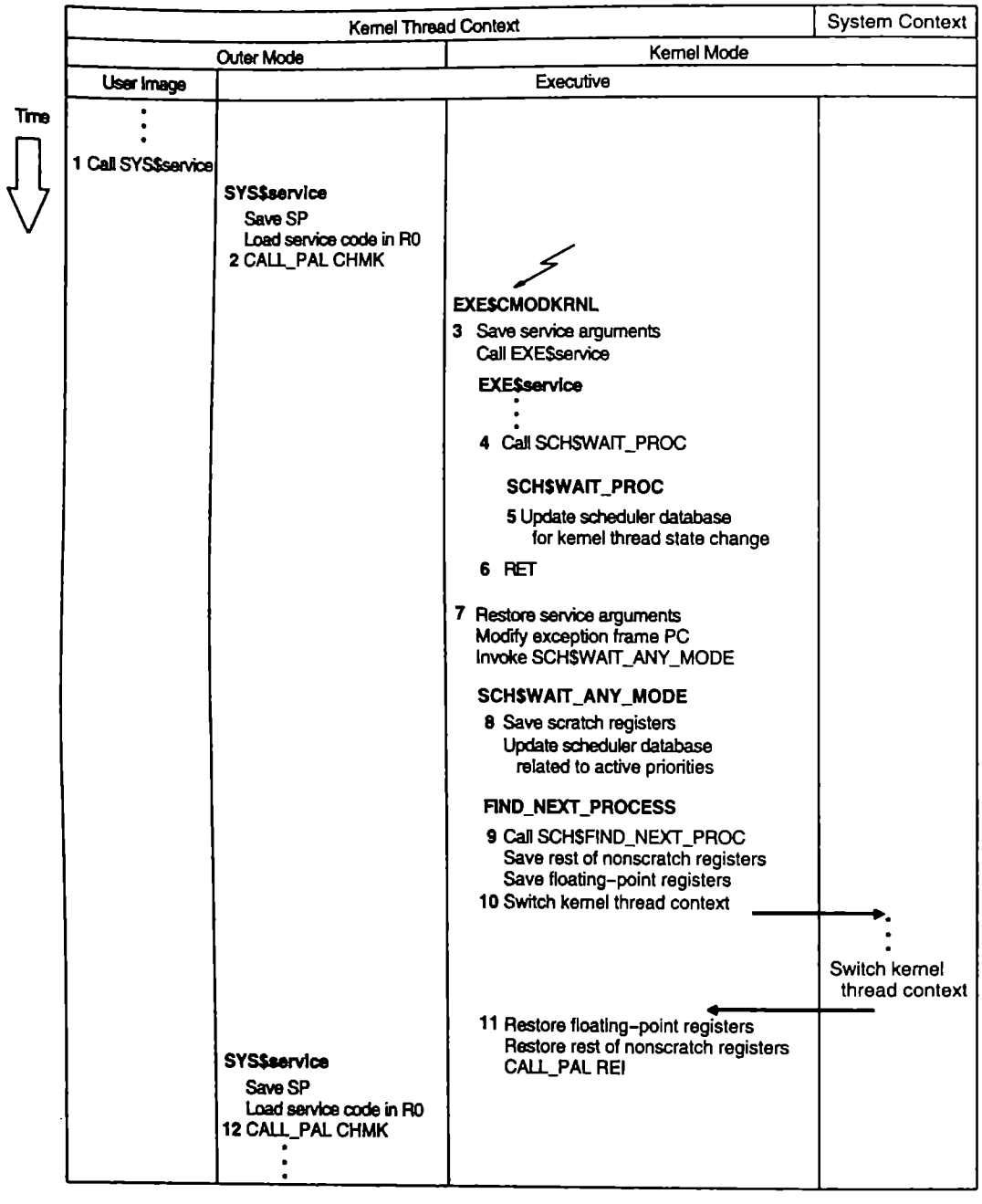



**Figure 2.13**
Kernel Thread Wait Sequence

a. It changes the scheduling state to that in the WQH$L_WQSTATE field of the specified wait queue header, inserts the KTB into the wait queue, and increments WQH$L_WQCNT to show the addition of a kernel thread to the queue.
b. It charges the SYSGEN parameter IOTA against the kernel thread quantum, as described in Section 2.6.2. It also adjusts PHD$L_TIMREF by the value of IOTA. PHD$L_TIMREF and the kernel thread quantum must be adjusted together for automatic working set limit adjustment to be responsive (see Chapter *Working Set List Dynamics*).
c. SCH$WAIT_PROC copies the contents of the system global EXE$GL_ABSTIM_TICS, the system time in soft ticks, to KTB$L_WAITIME, to record the time at which the kernel thread began its wait. If the kernel thread remains in a wait state for long, its process will become a candidate for working set shrinkage and possibly outswapping (see Chapter *The Swapper*).
d. It returns to the service procedure.

At this point, although the scheduler database shows the kernel thread as being in a wait state, the service procedure and the code described in the following steps continue to run in the kernel thread's hardware context until a `CALL_PAL SWPCTX` instruction is executed.

⑥ Still holding the SCHED spinlock, the service procedure returns the status SS$_WAIT_CALLERS_MODE to the change mode dispatcher.

⑦ In response to that status, the change mode dispatcher tests whether this is a service procedure that is to be reexecuted after the wait condition is satisfied. (Most service procedures that return this status must be reexecuted.) If so, it restores the argument registers (which may have been overwritten during service execution) from the kernel stack and subtracts 12 from the PC in the exception frame so that it contains the address of the beginning of the system service transfer routine.

In either case, the change mode dispatcher then calls SCH$WAIT_ANY_MODE, in module SCHEDULER. Chapter *System Service Dispatching* describes the change mode dispatcher and system service transfer routines in more detail.

⑧ SCH$WAIT_ANY_MODE is entered with the SCHED spinlock still held and nothing on the kernel stack except the CHMK exception stack frame. It takes the following steps:

a. It reserves space on the stack for all the scratch integer registers but saves only the system service argument registers. The contents of the others are irrelevant because the system service was entered through a procedure call, across which scratch integer register contents are never preserved.

b. It tests at what IPL the kernel thread is to be waited and, if it is less than 2, reads the kernel thread's ASTSR and ASTEN registers to determine whether a deliverable AST has been queued to the kernel thread but not yet delivered. This test prevents an AST event that should take the kernel thread out of its wait from being ignored. If a deliverable AST has been queued, SCH$WAIT_ANY_MODE reports an AST queuing event to SCH$REPORT_EVENT (see Section 2.6.5), which changes the kernel thread's scheduling state to COM.
c. It gets the address of the current kernel thread's KTB and its current priority from the per-CPU database.
d. It clears the bit representing the CPU's ID in the longword corresponding to the priority in the array at SCH$AL_CPU_PRIORITY.
e. If there are no other kernel threads at this priority current on any CPU, it clears the bit corresponding to that priority in SCH$GQ_ACTIVE_PRIORITY.
f. At this point, SCH$WAIT_ANY_MODE and the rescheduling interrupt service routine converge, at label FIND_NEXT_PROCESS.

⑨ The rescheduling interrupt service routine selects the next kernel thread to execute and saves the rest of the nonprivileged hardware context of the kernel thread being waited. The canonical kernel stack is now complete.
⑩ It executes the instruction `CALL_PAL SWPCTX`. Section 2.7.1 describes the selection of a new kernel thread and the context switch.
⑪ Later, when the kernel thread's wait is ended, it is placed back into execution in the outer mode at the beginning of the system service transfer routine.
⑫ The kernel thread reexecutes the transfer routine and reenters the service procedure.

Procedures that update the scheduler database to reflect a state change for a kernel thread being placed into a wait include

- SCH$WAIT_PROC, in module SCHED_ROUTINES, intended for a kernel thread to be placed into a specified wait in an outer mode
- SCH$RESOURCE_WAIT_SETUP, in module MUTEX, intended for a kernel thread to be placed into an MWAIT for a specified system resource to be available
- MMG$RESRCWAIT, in module PAGEFAULT, typically used for a page faulting kernel thread to be placed into an MWAIT for a specified system resource to be available
- MMG$PGFLTWAIT, in module PAGEFAULT, typically used for a page faulting kernel thread to be placed into a specified wait

The actions to complete a canonical kernel stack and initiate context swap are centralized in several routines in module SCHEDULER:

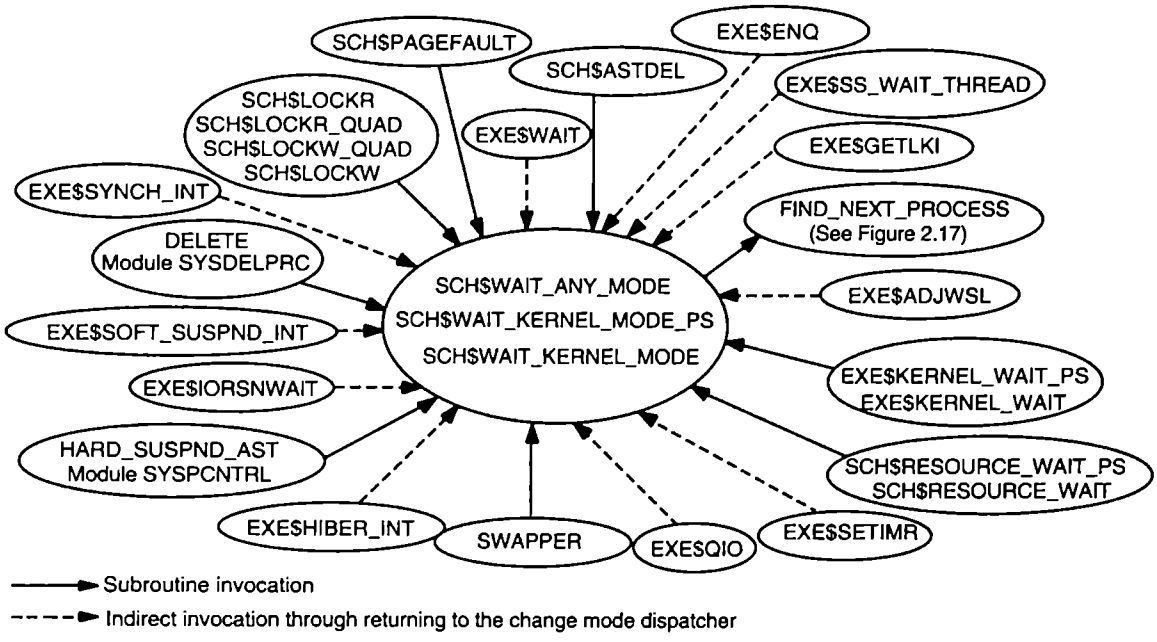



**Figure 2.14**
Paths Leading to a Kernel Thread Wait

- SCH$WAIT_ANY_MODE—Wait a kernel thread in the access mode specified by the PS in its interrupt/exception stack frame.
- SCH$WAIT_KERNEL_MODE—Wait a kernel thread in kernel mode using the current PS and caller's return address as the PC.
- SCH$WAIT_KERNEL_MODE_PS—Wait a kernel thread in kernel mode using a specified PS and caller's return address as the PC.

As shown in Figure 2.14, those routines are called by procedures such as the following:

- EXE$WAIT, in module SYSWAIT, to place a kernel thread requesting the $WAITFR, $WFLOR, or $WFLAND system service into an LEF or CEF wait (see Chapter 3)
- EXE$HIBER_INT, in module SYSPCNTRL, to place a kernel thread into a HIB wait (see Chapter 6)
- EXE$SOFT_SUSPND_INT and HARD_SUSPND_AST, in module SYSPCNTRL, to place a kernel thread into a SUSP wait (see Chapter 6)
- Various other system service procedures that return the status SS$_WAIT_CALLERS_MODE to the change mode dispatcher
- Swapper main loop, in module SWAPPER, to place the swapper into a HIB wait state when it has no work to do
- SCH$PAGEFAULT, in module PAGEFAULT, to place a kernel thread into PFW and other memory-management-related waits

- EXE$KERNEL_WAIT_PS, in module MUTEX, to place a kernel thread into an MWAIT for JIB byte count quota
- The various mutex lock routines, in module MUTEX, to place a kernel thread into an MWAIT for a mutex (see Chapter *Synchronization Techniques)*
- SCH$RESOURCE_WAIT and SCH$RESOURCE_WAIT_PS, in module MUTEX, to place a kernel thread into an MWAIT for a system resource
- SCH$ASTDEL, to place a kernel thread that needs the inner mode semaphore for a piggyback special kernel mode AST into an INNER_MODE MWAIT (see Chapter 6)
- EXE$SYNCH_INT, to place a kernel thread requesting the $SYNCH system service into an LEF or CEF wait (see Chapter 3)
- EXE$SS_WAIT_THREAD, to place a kernel thread that needs the inner mode semaphore to execute an executive or kernel mode system service into an INNER_MODE MWAIT (see Chapter 6)

One of the responsibilities of all the procedures that place a kernel thread into a wait is to ensure that the kernel thread can reenter the appropriate wait state, if necessary, after the kernel thread is placed back into execution as the result of AST delivery. (Recall that AST queuing makes a kernel thread in most wait states computable.) These routines therefore determine values for the PC and PS at which the kernel thread is to wait. These values are stored in the interrupt/exception stack frame, at the high-address end of the canonical kernel stack, that is used when the kernel thread executes a `CALL_PAL REI` instruction after being placed back into execution.

The PC and PS control which thread of execution will run, and its access mode and IPL. Its access mode affects AST delivery: only ASTs equally or more privileged can be delivered. If the access mode is kernel, then the wait IPL is also significant: an IPL of 2 blocks AST delivery interrupts. Several different techniques for placing a kernel thread into a wait are used, depending on the particular wait state being entered.

2.6.1.1 **Context for HIB Wait States.** When a kernel thread enters a HIB wait state, the change mode dispatcher uses the system service transfer routine CHMK exception PS as the wait PS. Consequently, the kernel thread waits in the access mode in which the system service was issued.

For the wait PC, the change mode dispatcher subtracts 12 from the CHMK exception PC in the exception stack frame so that it contains the address of the beginning of the system service transfer routine. Chapter *System Service Dispatching* contains more information about system service transfer routines and change mode exceptions.

If an AST is delivered to a kernel thread in such a wait state, when the AST exits, the AST delivery interrupt service routine's `CALL_PAL REI` returns control to the system service transfer routine.

2.6.1.2 **Context for LEF or CEF Wait State.** The OpenVMS Alpha event flag wait system services are mode of caller services. These services can check whether a local event flag is set without the overhead of changing to kernel mode. If the flag is clear, each service must request a kernel mode form of the service to recheck the state of the flag and wait the kernel thread if necessary.

When a kernel thread enters an LEF wait state, the change mode dispatcher does not modify the exception stack frame. Consequently, the kernel thread is waited in such a way that it will be placed back into execution within the mode of caller service at the outer mode, just after the request for the kernel mode form of the service.

Checking the state of common event flags requires kernel mode. When these services are requested to wait for common flags, they must each request a kernel mode form of the service to locate the flags, check them, and wait the kernel thread if necessary.

When a kernel thread enters a CEF wait state, the change mode dispatcher subtracts 12 from the PC in the exception stack frame so that when the kernel thread is placed back into execution, it will rerequest the kernel mode form of the service. Chapter 3 describes these services in more detail.

2.6.1.3 **Context for Memory Management Wait States.** SCH$PAGEFAULT, in module SCHEDULER, the page fault exception service routine (see Chapter *Paging Dynamics*), places kernel threads into the three wait states associated with memory management. There are other ways in which a kernel thread can be placed into these waits, but a page fault is the most common triggering event. SCH$PAGEFAULT uses the PC and PS in the page fault exception stack frame as the wait PC and PS. Because the PS reflects the access mode in which the page fault occurred, ASTs can be delivered for that and all inner access modes.

After an AST executes in such a kernel thread, the kernel thread executes the faulting instruction again. If the reason for the fault has been removed (a free page became available or the page read completed) while the AST was being delivered or was executing, the kernel thread simply continues with its execution. If the situation that caused the kernel thread to page fault still exists, the kernel thread reincurs the page fault and will be placed back into a memory management wait state. Note that a kernel thread that was initially in a PFW state for a shared page could be placed into a COLPG state by such a sequence of events. A kernel thread in a multithreaded process that was initially in a PFW state for a process-private page could be placed into a COLPG state.

2.6.1.4 **Context for a SUSP Wait.** A kernel thread is suspended as the result of executing an AST. The access mode of the AST can be supervisor or kernel mode, depending on which form of suspend is requested. The default is supervisor mode. While a kernel thread is suspended in kernel mode, the wait

PC is an address in the kernel AST that caused the kernel thread to enter the suspend state. The saved PS indicates kernel mode and IPL 0. ASTs can be queued, but not delivered, to a kernel thread suspended in kernel mode. That is, when an AST is queued to a kernel mode suspended kernel thread, the AST event is ignored.

While a kernel thread is suspended in supervisor mode, the saved PC is an address in the $SOFT_SUSPND_INT system service transfer routine, requested from within the supervisor mode AST, which remains active. AST queuing makes the kernel thread computable. When the kernel thread is placed into execution, a kernel or executive mode AST can be executed, but a user or supervisor mode AST cannot: the new AST control block remains queued, and the interrupt is dismissed. In either case, a `CALL_PAL REI` instruction is executed, which causes control to return to the wait PC. It repeats the system service request and repeats the test that suspended the kernel thread. If the kernel thread has not been resumed, it is suspended again.

2.6.1.5 **Context for an MWAIT Wait.** When a kernel thread is placed into a wait for a mutex, its saved PC is one of the lock routines in module MUTEX. Its saved PS indicates kernel mode and IPL 2, making it impossible for it to receive ASTs.

A kernel thread can also be placed into an MWAIT state while waiting for an arbitrary system resource. In this case, the caller of SCH$RESOURCE_WAIT or SCH$RESOURCE_WAIT_PS determines the wait PC and PS.

A kernel thread with resource wait mode enabled can be placed into an MWAIT state while waiting for a job quota, either byte count or TQE quota.

In the case of byte count, the routine EXE$DEBIT_BYTCNT, in module EXSUBROUT, checks whether the job has sufficient byte count quota for a particular request. If it does not, EXE$DEBIT_BYTCNT calls EXE$KERNEL_WAIT_PS to place the kernel thread into a wait with kernel access mode and IPL equal to that at entry to EXE$DEBIT_BYTCNT. Typically, this routine and its subentry points are called from device driver preprocessing routines at IPL 2, and thus the kernel thread is waited at IPL 2. The wait PC is an address within EXE$DEBIT_BYTCNT that repeats the test for sufficient byte count quota.

In the case of TQE quota, the kernel thread is placed into a wait similar to that for HIB and CEF—its wait PC is the address of the `CHMK` instruction in the system service transfer routine and its PS is the change mode exception PS, so that the kernel thread waits in the access mode from which it requested the service.

2.6.1.6 **Proactive Memory Reclamation from Periodically Waking Processes.** EXE$HIBER_INT and EXE$WAIT are responsible for implementing the policy of proactive memory reclamation from periodically waking processes.

The policy is enabled when the low bit of SYSGEN parameter MMG_CTLFLAGS is set, as it is by default. If enabled, the proactive memory reclamation mechanism becomes active only when memory is low. Each of the services checks whether the mechanism is active, whether the kernel thread being waited has a normal priority, and whether the process has accumulated 30 seconds of wait time (PCB$L_ACC_WAITIME) since the last time its execution history was checked. In the case of a multithreaded process, each kernel thread must have a normal priority and be in HIB, LEF, or CEF state. If all the conditions are met, each service procedure calls EXE$CHK_WAIT_BHVR, in module RSE.

EXE$CHK_WAIT_BHVR takes the following steps:

1. It checks whether the process has any outstanding direct I/O and, if so, returns immediately.
2. It checks whether the process has a high ratio of wait time to execution time since this routine was last called to check this process. If the process's accumulated CPU time is at least 1 percent of its wait time, the routine continues with step 5.
3. It tests whether the process has disabled automatic working set adjustment (PCB$V_DISAWS in PCB$L_STS is set) or whether the executive has temporarily blocked changes to its working set (PHD$V_NO_WS_CHNG in PHD$L_FLAGS is set). If either is true, EXE$CHK_WAIT_BHVR continues with step 5.
4. Holding the MMG spinlock, it tries to reduce the process's working set size by 25 percent. It does not alter the working set limit.
5. It copies the accumulated CPU time from PHD$L_CPUTIME to PCB$L_CPUTIME_REF for use the next time EXE$CHK_WAIT_BHVR is executed. It clears PCB$L_ACC_WAITIME and returns.

### 2.6.2 Quantum Expiration

The SYSGEN parameter QUANTUM defines the size of the time slice for the round-robin scheduling of normal kernel threads. The quantum also determines, for most scheduling states, the minimum amount of time a single-threaded process remains in memory after an inswap operation, but it is not an absolute guarantee of memory residence. The swapper's use of PCB$V_INQUAN in PCB$L_STS (the initial quantum flag) in selecting an outswap candidate is described in Chapter *The Swapper*. The value of QUANTUM is the number of 10-millisecond intervals (soft ticks) in the quantum. The default QUANTUM value of 20, therefore, produces a scheduling interval of 200 milliseconds.

A kernel thread's quantum is expressed as a negative number of soft ticks. After each 10-millisecond interval, the interval timer interrupt service routine increments the KTB$L_QUANT field in the current kernel thread's KTB. When this value becomes zero or positive, the interrupt service routine

requests a software timer interrupt. The software timer interrupt service routine signals a quantum-end event by invoking the subroutine SCH$QEND, in module RSE.

An additional deduction from quantum is governed by the special SYSGEN parameter IOTA. Its default value is 2, representing two soft ticks. This value is charged against KTB$L_QUANT each time a kernel thread enters a wait state. This mechanism ensures that all kernel threads experience quantum-end events with some regularity. Kernel threads that are compute-bound experience quantum end as a result of using a certain amount of CPU time. Kernel threads that are I/O-bound experience quantum end as a result of performing a reasonable number of I/O requests.

The routine SCH$QEND is executed whenever a current kernel thread reaches quantum end. It runs on the same CPU as the kernel thread, as part of the software timer interrupt service routine.

Its first action is to enable or disable alignment fault reporting for the current process (see Chapter *Condition Handling*). After acquiring the SCHED spinlock, its subsequent minimum actions are to reset the field KTB$L_QUANT to the full quantum value; clear the initial quantum flag, PCB$V_INQUAN in the field PCB$L_STS; and record EXE$GL_ABSTIM_TICS in KTB$L_ONQTIME. It performs those actions for both real-time and normal kernel threads.

For a normal kernel thread, SCH$QEND takes the following additional steps:

1. SCH$QEND updates PCB$L_PIXHIST, the pixscan history summary longword (see Section 2.6.7), by shifting it left one bit.
2. SCH$QEND tests whether a CPU time limit has been imposed on the process and, if so, compares the process's limit field, PHD$L_CPULIM, against its accumulated CPU time, PHD$L_CPUTIM, to determine whether that limit has been reached. If the CPU limit has been reached, SCH$QEND gives the process an interval of time to clean up or run down and queues an AST to the kernel thread. Initially the AST is a user mode AST that requests the $EXIT system service. If the process runs out of CPU time again, the next AST queued is supervisor mode, and so on. Each access mode has an interval of time for the process to clean up or run down before the image exits and the process is deleted. The size of the warning interval for each access mode is defined by the SYSGEN parameter EXTRACPU, which has a default value of 10 seconds.
3. SCH$QEND checks whether automatic working set limit adjustment is enabled (PCB$V_DISAWS in PCB$L_STS is clear) and appropriate for this process. If both are true, the limit of the process working set list may be expanded or contracted. Chapter *Working Set List Dynamics* describes automatic working set limit adjustment.

4. If the CLASS_SCHEDULER executive image is loaded (nonzero SCH$GL_TBSH) and if the kernel thread experiencing quantum end is not that of the null process, SCH$QEND calls SCH$CLASS_QEND, in module SYSSCHED, to notify it of the quantum-end event. If SCH$CLASS_QEND has set bit KTB$V_IN_TBS_STATE in KTB$L_STS2, SCH$QEND continues with step 7, forcing a reschedule. Section 2.8 discusses class scheduling.
5. If there is an inswap candidate (if SCH$GQ_COMOQS is nonzero, indicating at least one nonempty COMO state queue), SCH$QEND sets the current priority of the kernel thread to its base priority. It changes, as appropriate, CPU$L_CUR_PRI, SCH$AL_CPU_PRIORITY, and SCH$GQ_ACTIVE_PRIORITY.

   Furthermore, it calls SCH$SWPWAKE, in module RSE, to awaken the swapper. As a computable, resident, real-time kernel thread of software priority 16, the swapper is likely to be the next kernel thread scheduled.
6. SCH$QEND checks whether there is a COM kernel thread of equal or higher priority. If not, the current kernel thread will continue to execute. If its current priority is not equal to its base priority, SCH$QEND decrements its current priority, calling SCH$CHANGE_CUR_PRIORITY to make the appropriate changes to CPU$L_CUR_PRI, SCH$AL_CPU_PRIORITY, and SCH$GQ_ACTIVE_PRIORITY. This decrement is equivalent to the one made every time a kernel thread is placed into execution. SCH$QEND then releases the SCHED spinlock and returns to the software timer interrupt service routine.

   These actions avoid unnecessary context switches when this kernel thread continues to be the best candidate to execute.
7. If there is a COM kernel thread of equal or higher priority, SCH$QEND sets bit KTB$V_QUANTUM_RESCHED in KTB$L_STS2 and requests an IPL 3 rescheduling interrupt and returns. When the interrupt is granted, the current kernel thread will be taken out of execution and another selected to execute.

Figure 2.17 includes SCH$QEND as a requestor of a rescheduling interrupt.

### 2.6.3 Changing the Priority of a Current Kernel Thread

Several routines change the priority of a current kernel thread:

- SCH$QEND, when a normal kernel thread reaches quantum end and there is a COMO process
- SCH$QEND, when a normal kernel thread not yet at its base priority will continue to execute (see Section 2.6.2 for a description of quantum-end processing)

- EXE$SETPRI, in module SYSSETPRI, when a kernel thread requests the $SETPRI system service (see Chapter 6)
- The mutex lock routines in module MUTEX, when a normal kernel thread locks a mutex and gets a temporary boost to priority 16
- SCH$UNLOCK and SCH$UNLOCK_QUAD, in module MUTEX, when a normal kernel thread unlocks a mutex and has its priority restored (see Chapter *Synchronization Techniques* for information on locking and unlocking mutexes)
- MOVEFU, a kernel mode AST procedure in module SYSGETJPI, when the target kernel thread's original priority is restored after a boost (see Chapter 6)
- EXE$RESCHED, in module SYSPARPRC, when a kernel thread requests the $RESCHED system service to lower its priority to its base and request a rescheduling interrupt (see Chapter 6)

The actions to change the priority of a current kernel thread are centralized in the routine SCH$CHANGE_CUR_PRIORITY, in module RSE. All the routines in the previous list call SCH$CHANGE_CUR_PRIORITY. Figure 2.17 shows its callers.

SCH$CHANGE_CUR_PRIORITY is entered at IPL$_SCHED and with the SCHED spinlock held. It can run in system context, called from SCH$QEND; in the context of the kernel thread whose priority is changing; or in the context of a kernel thread requesting the $SETPRI service on behalf of another kernel thread. Its arguments specify the address of the KTB of the target kernel thread, the one whose priority is to be changed; the address of the per-CPU database of the CPU on which the target kernel thread is running; and the new priority.

SCH$CHANGE_CUR_PRIORITY takes the following steps:

1. It clears the bit corresponding to the CPU's ID in the longword corresponding to the priority in the array at SCH$AL_CPU_PRIORITY.
2. If there are no other kernel threads at this priority current on any CPU, it clears the bit corresponding to that priority in SCH$GQ_ACTIVE_PRIORITY.
3. It copies the new priority to KTB$L_PRI and CPU$L_CUR_PRI.
4. It sets the bit corresponding to the CPU's ID in the longword corresponding to the new priority in the array at SCH$AL_CPU_PRIORITY.
5. It sets the bit corresponding to the kernel thread's new priority in SCH$GQ_ACTIVE_PRIORITY.
6. It locates the least significant set bit in the quadword SCH$GQ_COMQS. The located bit position indicates the highest priority nonempty computable resident state queue.
7. It compares the changed priority of the target kernel thread with that of the highest priority COM kernel thread. If the changed priority is higher or equal, SCH$CHANGE_CUR_PRIORITY returns.

8. Otherwise, it requests a rescheduling interrupt on the CPU on which the target kernel thread is current.
   - If SCH$CHANGE_CUR_PRIORITY and the target kernel thread are executing on the same CPU, this is simply an IPL 3 software interrupt request.
   - If the CPUs are different, SCH$CHANGE_CUR_PRIORITY requests an interprocessor interrupt on the other CPU so that the IPL 3 interrupt can be requested there. Chapter *Synchronization Techniques* describes interprocessor interrupts.

Clearly, this priority comparison can result in rescheduling when the priority of a current kernel thread is lowered. Moreover, under some circumstances, it could also result in rescheduling even when the priority of a current kernel thread is raised a small amount. Because preemption of a current kernel thread by a newly computable kernel thread requires a priority difference of PRIORITY_OFFSET + 1, a normal computable kernel thread might continue to execute despite the existence of a slightly higher priority kernel thread that had just become computable. Under these circumstances, if the current kernel thread were to raise its priority to a value less than that of the newly computable process, the current process would be rescheduled.

### 2.6.4 Capability Mismatch

This section summarizes how a mismatch in capability requirements can occur between a current kernel thread and the available processors in an SMP system and how the mismatch is handled.

Several routines can affect kernel thread capability and affinity requirements. By default these routines defend against attempts to set requirements that cannot be met by the available processors.

SCH$REMOVE_CPU_CAP, in module CAPABILITY, is called to remove a capability from one or all CPUs. It takes the following steps, looping through them if all CPUs are to be affected:

1. It locks the CPU mutex to freeze active set membership (see Chapter *Symmetric Multiprocessing*).
2. It clears the bit corresponding to the capability in the target CPU's per-CPU database field CPU$L_CAPABILITIES and its longword in the SCH$AL_CPU_CAP array.
3. It gets the address of the kernel thread current on that CPU from CPU$L_CURKTB and examines its capability mask. If that kernel thread can no longer run, it requests a rescheduling interrupt, through an interprocessor interrupt if necessary.
4. It increments SCH$GL_CAPABILITY_SEQUENCE to indicate a change in the capabilities of the active members of the SMP system.
5. It releases the CPU mutex.

A kernel thread that requires a capability no longer present or that has explicit affinity for a single CPU that has been shut down is placed into an RWCAP MWAIT. Once in that state, the kernel thread can become computable only if another kernel thread changes the waiting thread's requirements or if the hardware configuration changes.

### 2.6.5 Event Reporting

This section describes how a kernel thread makes a transition to a COM state and how it preempts a current kernel thread.

A system event potentially changes the scheduling state of a kernel thread, making it computable, memory-resident, or outswapped. Examples of system events include the setting of an event flag for which a kernel thread is waiting, AST queuing, and page fault I/O completion. An executive routine aware of a system event that may take a kernel thread out of a wait state reports it on behalf of the affected kernel thread.

Holding the SCHED spinlock and running at IPL$_SCHED, such a routine invokes the RPTEVT macro, which generates the following code:

```
MOVL    #EVT$_event_name,R0
JSB     SCH$REPORT_EVENT
```

The event value identifies the event being reported.

SCH$REPORT_EVENT is responsible for effecting many of the kernel thread state transitions shown in Figure 2.12. Figure 2.15 shows the callers of SCH$REPORT_EVENT and its entry points SCH$CHSE and SCH$CHSEP.

SCH$REPORT_EVENT is also passed the address of the KTB of the affected kernel thread and a priority increment class. If the event makes the kernel thread computable, the kernel thread may receive a priority boost, depending on the priority class, its current priority, and its base priority.

SCH$REPORT_EVENT and routines it calls, all in module RSE, perform the following operations:

1. SCH$REPORT_EVENT indexes an internal table with the event value to determine whether the event is significant for the kernel thread, based on its current state.

   Each event has a bit mask defining which states this event can affect. The current state of the kernel thread is obtained from the KTB$L_STATE field.

   - For example, a wake event is only significant for kernel threads that are hibernating (HIB or HIBO states).
   - An outswap event is only significant for the four states (COM, HIB, LEF, and SUSP) where a wait queue change is required.
   - The queuing of an AST is significant for all kernel thread states except kernel mode SUSP and SUSPO, COM, COMO, and CUR, and results in a transition to COM or COMO.

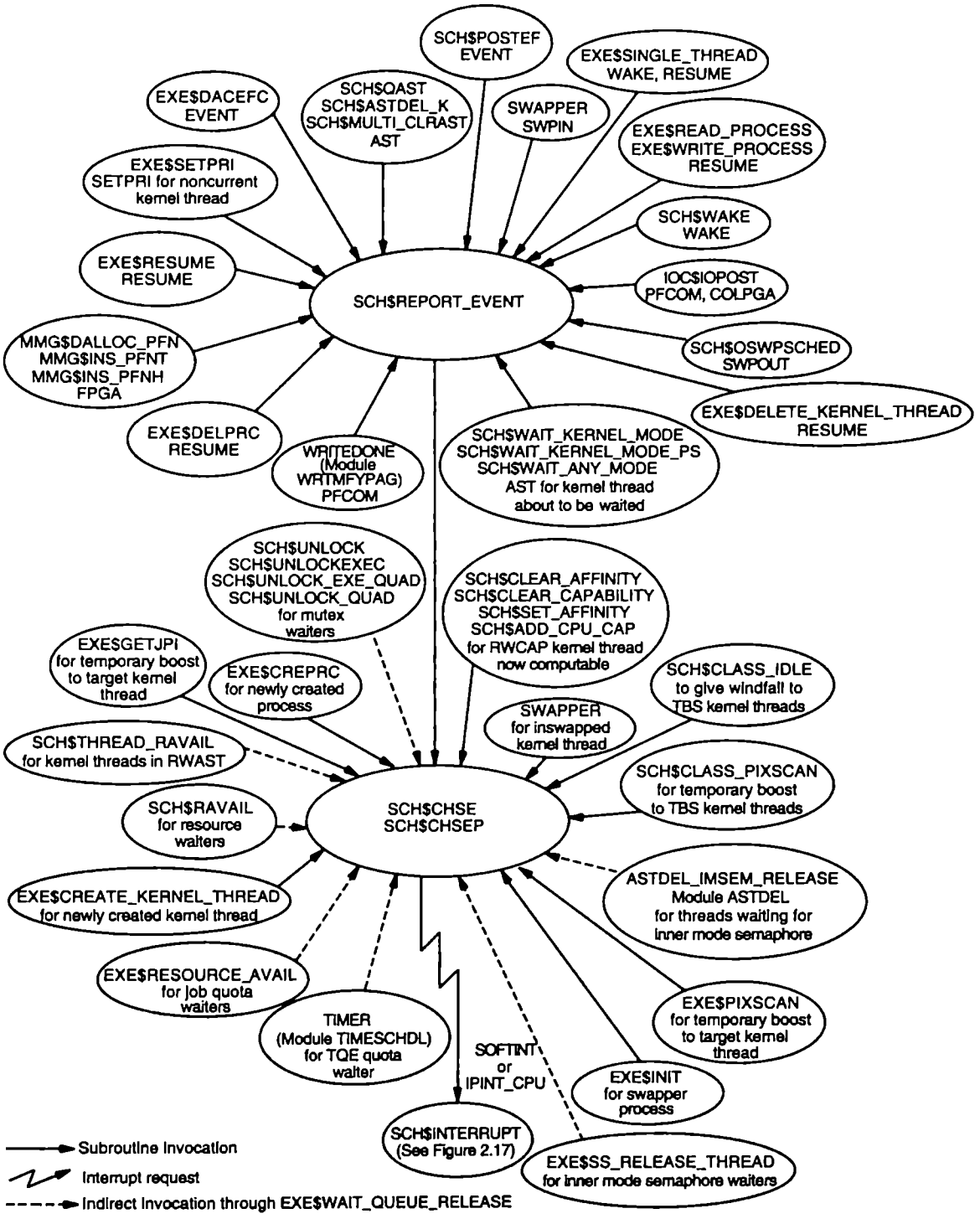



**Figure 2.15**
Paths to Event Reporting

2. If the event is not significant for the current kernel thread state, SCH$REPORT_EVENT ignores the event and simply returns.
3. For an outswap event producing an LEF to LEFO, HIB to HIBO, or SUSP to SUSPO transition, SCH$REPORT_EVENT removes the KTB of the kernel thread from the resident wait queue and inserts it into the corresponding outswapped wait queue. It adjusts the corresponding wait queue header count fields and KTBSL_STATE and returns.
4. For an outswap event producing a COM to COMO transition, SCH$REPORT_EVENT removes the KTB from the COM priority queue corresponding to KTBSL_PRI and inserts it into the corresponding COMO priority queue. It changes KTBSL_STATE to COMO. It clears the SCH$GQ_COMQS summary bit corresponding to KTBSL_PRI if that COM queue is now empty and unconditionally sets the corresponding SCH$GQ_COMOQS bit. It then returns.
5. For any transition that makes a kernel thread computable, SCH$REPORT_EVENT subtracts KTBSL_WAITIME from the current time in soft ticks and adds it to KTBSL_ONQTIME to subtract out from it the effect of the time spent waiting. It also adds the result to the process's accumulated wait time, PCBSL_ACC_WAITIME.

   It decrements the wait queue header count of whichever wait queue the kernel thread is in.
6. SCH$REPORT_EVENT falls through to SCH$CHSE and SCH$CHSEP. SCH$CHSE performs whatever priority adjustment is appropriate (see Section 2.6.6) and removes the kernel thread from whatever queue it is in.
7. If the now computable kernel thread is in a class-scheduled process that is out of quota (KTBSV_IN_TBS_STATE in KTBSL_STS2 is set), SCH$CHSEP takes the steps described in Section 2.8 and returns.
8. If the now computable kernel thread is outswapped, SCH$CHSEP changes its state to COMO and inserts the kernel thread into the COMO queue corresponding to its priority, and unconditionally sets the SCH$GQ_COMOQS summary bit corresponding to the selected priority queue. It awakens the swapper and returns. Later, after the kernel thread is inswapped, it will become eligible for execution.
9. If the now computable kernel thread is resident, SCH$CHSEP changes its state to COM, inserts the kernel thread into the COM queue corresponding to its priority, and unconditionally sets the SCH$GL_COMQS summary bit corresponding to the selected priority queue.

   It compares the kernel thread's current affinity mask with the mask of idle CPUs. If there are potential CPUs on which the kernel thread can execute, SCH$CHSEP clears their bits in SCH$GL_IDLE_CPUS as a signal to each of them to try to reschedule (see Section 2.7.1). If it appears that there are no potential CPUs, SCH$CHSEP checks that the kernel thread's KTBSL_CAPABILITY_SEQ is current, recalculating cur-

rent affinity if it is not. If there are still no idle candidate CPUs and the kernel thread's priority is not high enough for it to preempt any active kernel thread, SCH$CHSEP simply returns.

If the kernel thread's priority permits it to preempt some active kernel threads, SCH$CHSEP searches for a candidate to preempt on a CPU whose capabilities fit. If it finds one, it requests either an interprocessor interrupt or an IPL 3 interrupt, depending on where the kernel thread to be preempted is executing. When the interprocessor interrupt is granted, its service routine will request an IPL 3 interrupt to cause rescheduling.

On a uniprocessor system, the issue is simpler: if there is a current kernel thread, can it be preempted by the newly computable kernel thread? The preemption test is based upon the SCH$AQ_PREEMPT_MASK array, described in Section 2.3. If there is no current kernel thread or if it can be preempted by the newly computable one, SCH$CHSEP requests an IPL 3 software interrupt and then returns.

### 2.6.6 System Events and Associated Priority Boosts

System routines that report events to SCH$REPORT_EVENT not only describe the event and the kernel thread to which it applies but also specify one of five classes of priority increments or boosts that may be applied to the base priority of the kernel thread. SCH$REPORT_EVENT uses the class as a longword index into a table of values (local symbol B_PINC in module RSE) to compute the actual boost. Table 2.3 lists the events, priority class, and potential amount of priority increment applied to the kernel thread. The table does not show AST queuing, because system routines queuing ASTs to a kernel thread can select any of the priority increment classes to be associated with the queuing of an AST.

The actual software priority of the kernel thread is determined by the following steps:

1. The priority boost for the event class (see Table 2.3) is added to the base priority of the kernel thread (KTB$L_PRIB).
2. If the kernel thread has a current priority higher than the result of step 1, the current priority is retained (as occurs in Figure 2.16, event 13).
3. If the higher priority of steps 1 and 2 is more than 15, then the base priority of the kernel thread is used. Note that this test accomplishes two checks at the same time. First, all real-time kernel threads fit this criterion, with the result that real-time kernel threads do not have their priorities adjusted in response to system events. Second, priority boosts cannot move a normal kernel thread into the real-time priority range.

A side effect of step 3 is that real-time kernel threads always execute at their base priorities. Further, normal kernel threads with base priorities from 10 to 15 do not always receive priority increments as events occur. As the

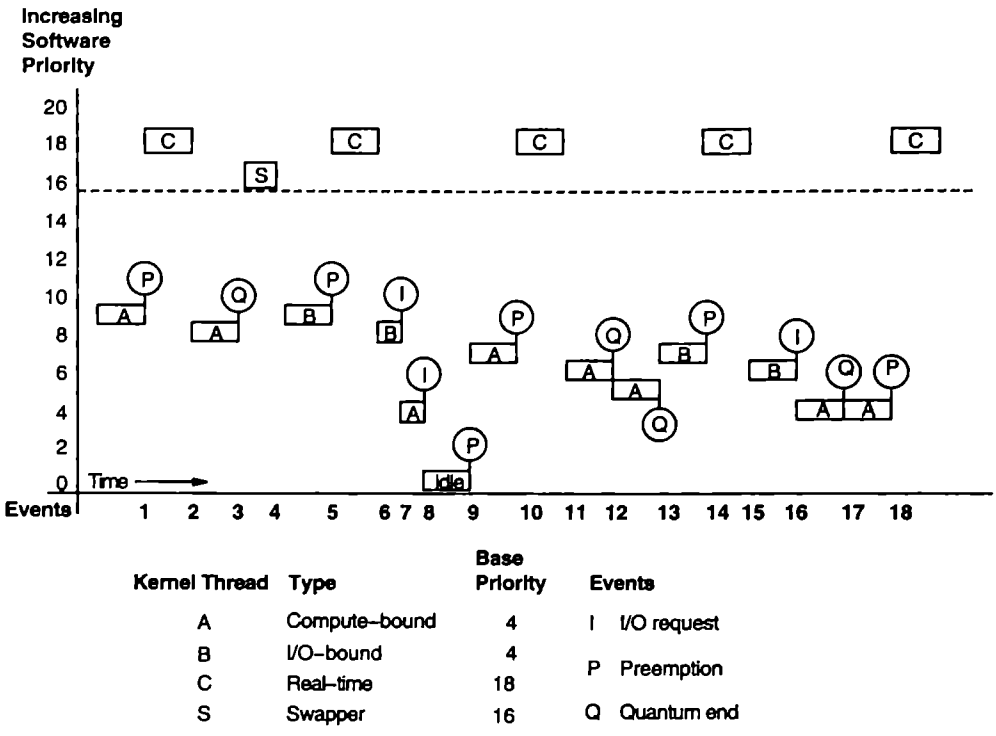
Table 2.3 System Events and Associated Priority Boosts

| |
|---|
| *Priority Class 0 (PRI$_NULL) and Priority Boost of 0* |
| Page fault read complete |
| Inswap |
| Outswap |
| Collided page available |
| Quantum end[1] |
| $GETxxI completion[2] |
| $SNDJBC completion[2] |
| Resume a kernel thread stalled during single thread operation |
| *Priority Class 1 (PRI$_IOCOM) and Priority Boost of 2* |
| Direct I/O completion[3] |
| Nonterminal buffered I/O completion[3] |
| Update section write completion[2] |
| Set priority |
| Event flag set through $SETEF |
| Modified write of deleted page complete |
| *Priority Class 2 (PRI$_RESAVL) and Priority Boost of 3* |
| Resource available |
| Mutex available |
| Job quota returned |
| Free page available |
| Resource lock granted[2] |
| Wake a kernel thread |
| Resume a kernel thread |
| Resume a kernel thread for deletion |
| Inner mode AST intended for this kernel thread had to be delivered to the kernel thread holding the inner mode semaphore (see Chapter 5) |
| Common event flag cluster being disassociated for whose flags kernel thread is waiting |
| Wake a kernel thread to acknowledge single thread request |
| *Priority Class 3 (PRI$_TOCOM) and Priority Boost of 4* |
| Terminal output completion[3] |
| *Priority Class 4 (PRI$_TICOM) and Priority Boost of 6* |
| Terminal input completion[3] |
| Process creation |
| *Priority Class 5 (PRI$_TIMER) and Priority Boost of 3* |
| Timer request expiration[2] |

[1] When a normal kernel thread reaches quantum end, its priority is lowered to its base if there is a COMO kernel thread. Otherwise, the kernel thread's priority is decremented by 1.

[2] This priority boost is part of reporting that the event flag associated with the request has been set. An AST may be queued to the kernel thread as well, with the same boost specified. The kernel thread priority is affected only if the kernel thread is in a wait.

[3] This priority boost is part of queuing an AST to the kernel thread.




| Kernel Thread | Type | Base Priority | Events |
|---|---|---|---|
| A | Compute-bound | 4 | I I/O request |
| B | I/O-bound | 4 | P Preemption |
| C | Real-time | 18 | |
| S | Swapper | 16 | Q Quantum end |

**Figure 2.16**
Priorities and Priority Adjustments

base priority of a normal kernel thread is moved closer to 15, the kernel thread spends a greater amount of time at its base priority. Priority 14 and 15 kernel threads receive no priority boosts. This strategy benefits those kernel threads that most need it—I/O-bound and interactive kernel threads with base priorities of 4 through 9. Kernel threads with elevated base priorities do not need this help because they are always at these levels.

An example of priority adjustment that occurs over time for several kernel threads is given in Figure 2.16. In this example, the value of the SYSGEN parameter PRIORITY_OFFSET is assumed to be 2. The following notes relate to the event numbers along the time axis of the figure:

1. Kernel thread C becomes computable. Kernel thread A is preempted.
2. C hibernates. A executes again, one priority level lower.
3. A experiences quantum end. Because there is a computable outswapped kernel thread (which is B), A's priority is lowered to its base. A continues to run.
4. The swapper now executes to complete its inswap of B, previously initiated, and B is scheduled for execution.
5. B is preempted by C.
6. B executes again, one priority level lower.

7. B requests an I/O operation to a device other than a terminal. A executes at its base priority.
8. A requests a terminal output operation and waits for its completion. There is no kernel thread that can be scheduled. The idle loop (see Section 2.7.3) executes.
9. A executes following I/O completion at its base priority + 3. (The applied boost was 4, and A's priority was subsequently decremented when it was rescheduled.)
10. A is preempted by C.
11. A executes again, one priority level lower.
12. A experiences quantum end, and its priority is lowered by 1. It continues to execute. A's priority is not lowered to its base because there is no computable outswapped kernel thread.
13. B's output completes. A priority boost of 2 is not applied to B's base priority because the result would be less than B's current priority. Although B's priority is higher than that of A, it is not high enough to preempt A, which continues to execute until quantum end. B then executes.
14. B is preempted by C.
15. B executes again, one priority level lower.
16. B requests an I/O operation. A executes again, one priority level lower. (A has reached its base priority.)
17. A experiences quantum end, and because there are no other computable kernel threads of equal or higher priority, A continues to execute at the same priority (its base priority).
18. A is preempted by C.

### 2.6.7 PIXSCAN Priority Boosts

The pixscan mechanism gives occasional priority boosts to normal priority COM and COMO kernel threads. The SYSGEN parameter PIXSCAN specifies the maximum number of kernel threads that can receive this boost each second. The priority boost prevents a high-priority, compute-intensive kernel thread from continuously blocking lower priority kernel threads and causing potential deadlocks. A deadlock might occur, for example, if a low-priority kernel thread acquired a volume lock on a critical disk but could not receive enough CPU time to complete its use of the lock and release it.

The mechanism is implemented in the routine SCH$PIXSCAN, in module RSE, called once a second from EXE$TIMEOUT (see Chapter *Time Support*).

SCH$PIXSCAN takes the following steps:

1. It first tests whether SGN$GW_PIXSCAN, the SYSGEN parameter, is 0. A zero value disables this mechanism, and SCH$PIXSCAN simply returns to its caller. The default value is 1.
2. A nonzero value in SGN$GW_PIXSCAN is the maximum number of kernel threads that may be boosted. SCH$PIXSCAN acquires the

SCHED spinlock. No IPL change is necessary because it is already executing at IPL 8.

3. It tests whether the CLASS_SCHEDULER executive image is loaded (nonzero contents of SCH$GL_TBSH). If so, SCH$PIXSCAN calls SCH$CLASS_PIXSCAN, in module SYSSCHED (see Section 2.8).
4. SCH$PIXSCAN determines whether any kernel threads are eligible for boost, that is, COM and COMO kernel threads with external priorities 0 through 15. If there are none, it releases the SCHED spinlock and returns.
5. If there are eligible kernel threads, it determines the priority of the highest priority normal kernel thread that is CUR, COM, or COMO. This is the value to which selected kernel threads will be boosted.
6. SCH$PIXSCAN uses the low bit of EXE$GL_ABSTIM as a "coin" to determine whether to begin scanning each priority level's compute queues with the COM or COMO queue. In an outer loop, it scans the COM and COMO queues, starting with the (external) priority 0. SCH$PIXSCAN stops when one of the following occurs:

   - It reaches the queues with the same priority as the boost value computed in step 5.
   - It has boosted the maximum number of kernel threads.
   - It reaches a kernel thread that has reached quantum end within a time interval less than the SYSGEN parameter DORMANTWAIT.

   Examining the kernel threads in a particular nonempty compute queue, SCH$PIXSCAN performs the following steps for each kernel thread:

   a. It compares KTB$L_ONQTIME plus the SYSGEN parameter DORMANTWAIT, expressed in 10-millisecond units, to the current absolute time, EXE$GL_ABSTIM_TICS. If the latter is less, the kernel thread is not dormant and has not been waiting for the CPU long enough to get a boost. By implication, no other kernel thread in that or any higher priority queue is likely to be dormant. The default value of DORMANTWAIT is 2 seconds.
   b. If the kernel thread is dormant, it calls SCH$CHSEP, in module RSE, to boost the kernel thread's priority. SCH$PIXSCAN sets the low-order bit in its process's pixscan history longword, PCB$L_PIXHIST. This longword is shifted left at each quantum end to record whether a kernel thread had a pixscan boost during its past executions. The pixscan history of a process is significant for quantum-end automatic working set limit reductions, as described in Chapter *Working Set List Dynamics*.

7. SCH$PIXSCAN releases the SCHED spinlock and returns.

## 2.7 RESCHEDULING INTERRUPT

The IPL 3 interrupt service routine schedules kernel threads for execution. The function of this interrupt service routine is to remove the currently executing kernel thread by saving the current kernel thread's hardware registers and replacing them with those of the highest priority computable resident kernel thread. This operation, known as context switching, is accompanied by modifications to the kernel thread state, current priority, and state queue of the affected kernel threads.

In some cases, a rescheduling interrupt is requested because a resident kernel thread has become computable whose priority is high enough to preempt the current kernel thread. In other cases, the current kernel thread enters a wait state, and the executive transfers into the rescheduling interrupt service routine to select another kernel thread to run. If there is no computable resident kernel thread, the interrupt service routine switches to the hardware context called the system HWPCB (see Section 2.1.5).

The Alpha architecture was designed so that it could be extended to assist the software in performing critical, commonly performed operations, such as context switching; it provides the instruction `CALL_PAL SWPCTX` to switch two kernel threads' privileged hardware contexts.

### 2.7.1 Rescheduling Interrupt Service Routine

The IPL 3 interrupt service routine consists of a central MACRO-64 interrupt service routine, SCH$INTERRUPT in module SCHEDULER, and several MACRO-32 routines.

As shown in Figure 2.17, SCH$INTERRUPT is requested as an IPL 3 software interrupt by several different routines:

- SCH$REPORT_EVENT and SCH$CHSE/SCH$CHSEP, when a resident kernel thread becomes computable whose priority allows it to preempt the current kernel thread
- SCH$QEND, when a current kernel thread reaches quantum end, it is a normal kernel thread, and there is a COM kernel thread of equal or higher priority
- SCH$CHANGE_CUR_PRIORITY, when a current kernel thread changes its priority and there is a COM kernel thread whose priority is higher
- SCH$ACQUIRE_IMPLICIT_AFFINITY, when a current kernel thread acquires implicit affinity for a processor other than the one on which it is executing
- SCH$REMOVE_CPU_CAP, when a current kernel thread is executing on a CPU that just lost a capability required by the kernel thread
- SCH$SET_CAPABILITY, when a current kernel thread requires a capability not present on the CPU on which it is executing
- SCH$SET_AFFINITY, when a current kernel thread sets explicit affinity for a CPU on which it is not running

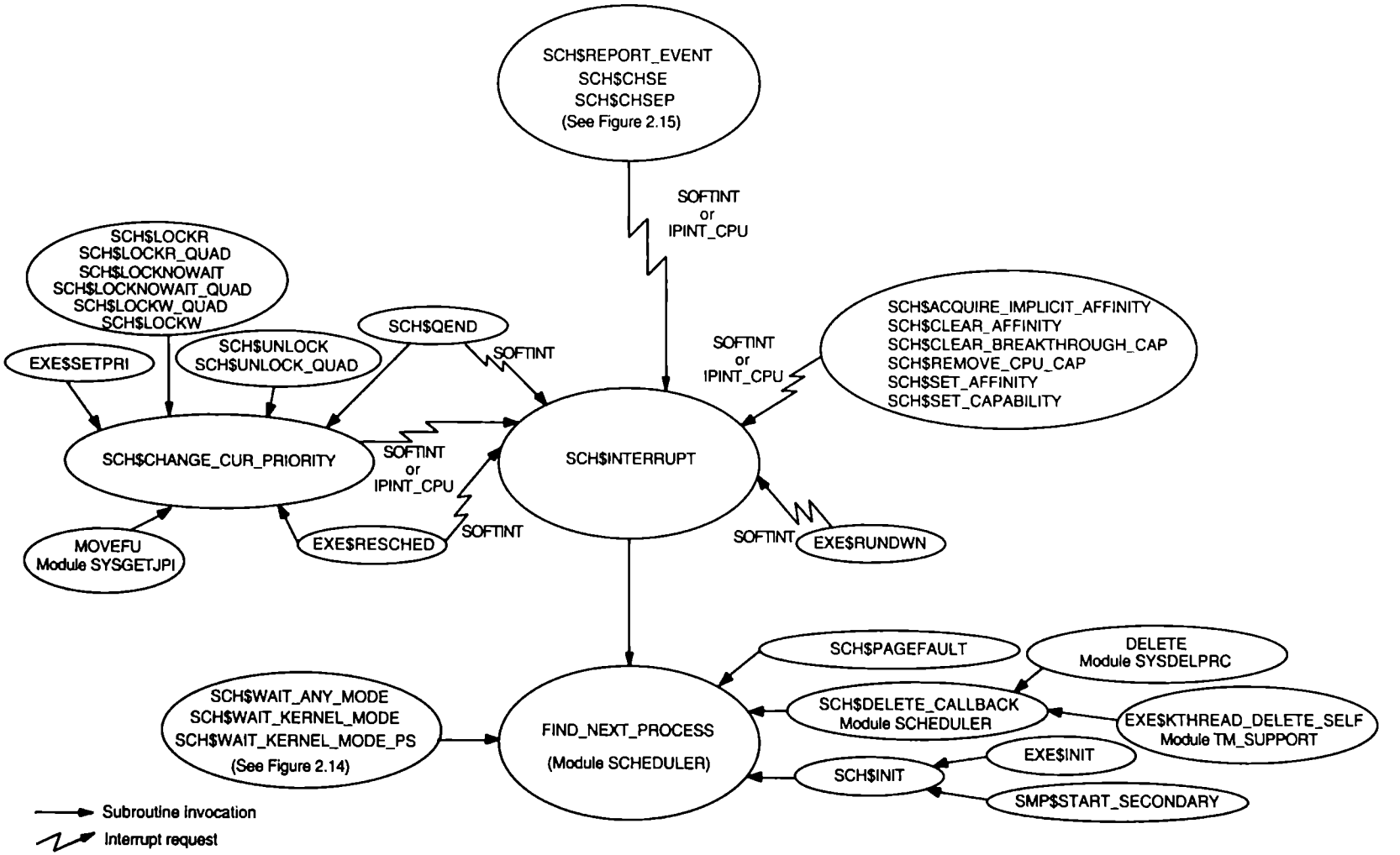



**Figure 2.17**
Paths to Rescheduling

- SCHSCLEAR_AFFINITY, when a current kernel thread changes its explicit affinities and this CPU is no longer one for which it has affinity
- SCHSCLEAR_BREAKTHROUGH_CAP, when a current kernel thread restores its saved capabilities and affinities and thus can no longer run on the current CPU
- EXESRUNDWN, when an initial kernel thread's just-restored permanent capability and affinity requirements do not match the capabilities of the CPU on which it is executing
- EXESRESCHED, in response to a kernel thread's explicit request for rescheduling

Note that sometimes an IPL 3 interrupt can be directly requested on the appropriate CPU. At other times, an interprocessor interrupt must be requested first so that the IPL 3 interrupt can be requested on the appropriate CPU by the interprocessor interrupt service routine.

Under some circumstances, there may not be a current kernel thread to be saved by SCH$INTERRUPT. In these cases, executive routines transfer control directly into the middle of SCH$INTERRUPT, at FIND_NEXT_PROCESS, for kernel thread selection.

The routines that transfer into the middle of SCH$INTERRUPT include the following:

- SCH$WAIT_ANY_MODE, SCH$WAIT_KERNEL_MODE, and SCH$WAIT_KERNEL_MODE_PS, when a current kernel thread has been placed into a wait state
- DELETE, in module SYSDELPRC, when a process's initial kernel thread has been deleted, through MACRO-64 routine SCH$DELETE_CALLBACK, in module SCHEDULER
- EXE$KTHREAD_DELETE_SELF, in module TM_SUPPORT, when a current kernel thread has been deleted, through SCH$DELETE_CALLBACK
- EXE$INIT, in module INIT, leaving system context during system initialization, to schedule the first kernel thread on the primary processor (or only processor), through MACRO-64 routine SCH$INIT, in module SCHEDULER
- SMP$START_SECONDARY, the routine that performs secondary processor initialization, leaving system context on a secondary processor, to schedule its first kernel thread, through SCH$INIT
- SCH$PAGEFAULT, when a current kernel thread has been placed in a wait state as a result of a page fault

SCH$INTERRUPT runs in kernel mode, initially on the kernel stack of the kernel thread being taken out of execution. The first part of the canonical kernel stack, the interrupt/exception stack frame, is already present. It performs the following steps:

1. SCH$INTERRUPT first saves all the scratch registers (R0, R1, R16–R28) on the kernel stack so that it can call other procedures.
2. It calls SCH$STATE_TO_COM, in module SCHED_ROUTINES, to update the scheduler database for the current kernel thread.
   SCH$STATE_TO_COM proceeds as follows:

   a. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED to block concurrent access to and modification of the scheduler database.
   b. It gets the address of the current kernel thread's KTB and its current priority from the per-CPU database.
   c. It clears the bit representing the CPU's ID in the longword corresponding to the priority in the array at SCH$AL_CPU_PRIORITY.
   d. If there are no other kernel threads at this priority current on any CPU, it clears the bit corresponding to that priority in SCH$GQ_ACTIVE_PRIORITY.
   e. SCH$STATE_TO_COM sets the bit corresponding to the kernel thread's priority in the compute queue summary quadword, SCH$GQ_COMQS.
   f. It changes the state of the kernel thread from CUR to COM by updating the KTB$L_STATE field.
   g. It inserts the KTB at the tail of the COM queue corresponding to the kernel thread's current priority.
   h. It clears SCH$GL_IDLE_CPUS to signal any idle CPU that it should attempt to reschedule. The scheduler idle loop is described in Section 2.7.3.
   i. SCH$STATE_TO_COM returns to SCH$INTERRUPT. At this point, although the scheduler database shows the kernel thread as COM, SCH$INTERRUPT is still running in its hardware context.

3. Still holding the SCHED spinlock, SCH$INTERRUPT calls SCH$FIND_NEXT_PROC, in module SCHED_ROUTINES, to select the next kernel thread to execute on this CPU. SCH$FIND_NEXT_PROC takes the following steps:

   a. It scans SCH$GQ_COMQS to find the least significant bit set. The located bit position indicates the highest priority nonempty computable resident state queue. As an optimization, it first tests whether there are any computable resident kernel threads and, if not, returns immediately. If there are, it uses a mask to test whether there are any at priorities 0 through 46. Typically, there are none, and the scan begins at bit 47 of SCH$GQ_COMQS, saving 47 fruitless scan iterations. Bit 47 corresponds to external priority 16, the priority of the swapper, which is often the highest priority computable kernel thread.

b. It uses the bit number as an index into the COM listheads to get the address of the listhead of the selected computable resident queue.
c. It removes the first KTB in the selected queue.
d. If the removed KTB was the only one in the queue, SCH$FIND_NEXT_PROC clears the corresponding SCH$GQ_COMQS bit to indicate that the queue is empty.
e. It confirms that the structure removed from the queue is actually a KTB, generating the fatal bugcheck INCON_SCHED if not.
f. SCH$FIND_NEXT_PROC tests whether the kernel thread's capability requirements and explicit affinities, if any, match the capabilities of the CPU. If they do not match, it tests further to see if the requirements can be met by any active SMP member. If they cannot, it places the kernel thread into a RWCAP resource wait state and selects another kernel thread to run. If the kernel thread has implicit affinity for a different CPU, SCH$FIND_NEXT_PROC tries to honor it but may not (see Section 2.5).
g. If the capabilities match, it stores the address of the new current kernel thread KTB in the per-CPU database.
h. It changes the state of the kernel thread to current by storing the value SCH$C_CUR in the KTB$L_STATE field.
i. It stores the CPU's ID in KTB$L_CPU_ID.
j. It examines the current kernel thread priority and potentially modifies it. If the kernel thread is real-time or a normal kernel thread already at its base priority, then the kernel thread is scheduled at its current or base priority (they are the same). If the current kernel thread is a normal kernel thread above its base priority, then a decrease of one software priority level is performed before scheduling. Thus, priority demotions always occur before execution, and a kernel thread executes at the priority of the queue to which it will be returned (not at the priority of the queue from which it was removed).
k. SCH$FIND_NEXT_PROC copies the kernel thread's current priority to the per-CPU database.
l. It clears the bit corresponding to this CPU in SCH$GL_IDLE_CPUS to indicate that the CPU is not idle.
m. It sets the bit corresponding to the CPU's ID in the longword corresponding to the priority in the array at SCH$AL_CPU_PRIORITY.
n. It sets the bit corresponding to the kernel thread's current priority in SCH$GQ_ACTIVE_PRIORITY.
o. Still holding the SCHED spinlock, it returns to SCH$INTERRUPT with the address of the KTB or, if it found no computable kernel thread that could run on this CPU, the value 0.

4. If there is no computable resident kernel thread, SCH$INTERRUPT executes the idle loop code, described in Section 2.7.3.

5. Before saving the rest of the current kernel thread's hardware context, SCH$INTERRUPT checks whether the current kernel thread is the same as the newly selected kernel thread. If so, it releases the SCHED spinlock and continues with step 19.
6. If they differ, it saves on the kernel stack the nonscratch registers whose contents have not already been saved (R8–R15, R29). This completes the canonical kernel stack.
7. If the current and newly selected kernel threads are not within the same process, SCH$INTERRUPT clears the bit corresponding to the CPU ID in the current PCB$L_ACTIVE_CPUS to indicate that this process no longer has a kernel thread current on this CPU. It decrements the current PCB$Q_ONCPUCNT to indicate one less current kernel thread.
8. If PCB$Q_ONCPUCNT is now zero, indicating no current kernel threads of this process, SCH$INTERRUPT saves the current kernel thread's ASN and the sequence number associated with it in PCB$Q_PRVASN and PCB$Q_PRVASNSEQ and the CPU ID in PCB$L_PRVCPU. This information will be used to determine whether a new ASN must be assigned when a kernel thread of that process is next placed into execution.
9. If the kernel thread is performing floating-point arithmetic, SCH$INTERRUPT saves the contents of the floating-point registers (F0–F30) and the FPCR in the FRED.
10. It sets the bit corresponding to the CPU ID in the new PCB$L_ACTIVE_CPUS to indicate a current kernel thread of that process on this CPU and increments PCB$Q_ONCPUCNT.
11. It then tests whether the newly selected kernel thread needs to be assigned a new ASN. If the process has only this current kernel thread, if the last kernel thread taken out of execution ran on this same processor, and if CPU$Q_ASNSEQ has not changed since the kernel thread was taken out of execution, SCH$INTERRUPT copies PCB$Q_PRVASN and PCB$Q_PRVASNSEQ to the new FRED$Q_ASN and FRED$Q_ASNSEQ. It continues with step 12.

    If any of those tests fails, SCH$INTERRUPT decrements the current contents of CPU$Q_ASN as the new ASN. When decrementing the number past zero, SCH$INTERRUPT takes the following additional steps:

    a. SCH$INTERRUPT resets CPU$Q_ASN to the maximum possible number on this type of processor and increments CPU$Q_ASNSEQ, the sequence number to be associated with the next uses of the address space numbers.
    b. Any process-private TB entries based on the old uses of the ASNs are stale and must be flushed. SCH$INTERRUPT executes the

instruction CALL_PAL MTPR_TBIAP to flush all TB entries whose address space match bits are clear.

It stores the ASN in the newly selected kernel thread's FRED$Q_ASN and in CPU$Q_ASN. It stores CPU$Q_ASNSEQ in FRED$Q_ASNSEQ.

12. It copies the physical address of the HWPCB for the scheduled kernel thread from KTB$L_PHYPCB to R16.
13. It executes a CALL_PAL SWPCTX instruction (see Section 2.7.2) to switch the privileged part of the two hardware contexts. SCH$INTERRUPT is now running in the context of the newly scheduled kernel thread and on its kernel stack.
14. If the newly scheduled kernel thread is performing floating-point arithmetic, SCH$INTERRUPT restores the contents of the floating-point registers (F0–F30) and the FPCR from the FRED.
15. SCH$INTERRUPT restores the kernel thread's nonscratch registers from its canonical kernel stack, leaving the scratch registers and interrupt/exception stack frame.
16. SCH$INTERRUPT tests and clears the FRED$V_AST_PENDING flag in FRED$L_FLAGS. It releases the SCHED spinlock. If the flag was clear, it continues with step 19.
17. If the flag was set, indicating that a deliverable AST has been queued to the kernel thread, SCH$INTERRUPT examines the PS in the interrupt/exception stack frame to determine at what mode and IPL the kernel thread will resume execution. If IPL is 2 or higher, no AST delivery is possible, and SCH$INTERRUPT continues with step 19.

    If IPL is lower than 2, SCH$INTERRUPT examines FRED$Q_ASTSR_ASTEN to get the contents of the kernel thread's AST summary and enable registers. It determines the mode of the most privileged pending AST. It checks whether the kernel thread is to resume execution at that or a less privileged mode and whether delivery to that mode is enabled. If either condition is false, it continues with step 19.
18. SCH$INTERRUPT has determined that the pending AST is deliverable. If the routine were to execute a CALL_PAL REI instruction now, an AST delivery interrupt would be requested and granted. To save some of the overhead associated with those steps, SCH$INTERRUPT instead clears the appropriate ASTSR bit and simulates the interrupt:

    a. It restores the scratch integer registers from the canonical kernel stack.
    b. It builds an interrupt stack frame whose PC is the address of the appropriate AST delivery interrupt service routine, whose PS specifies a current mode of kernel and IPL 2, and whose R3 is the linkage pointer for the service routine.

c. It executes a CALL_PAL REI instruction, transferring control to the AST delivery interrupt service routine. After the AST procedure exits, the kernel thread will resume execution at the place at which it was taken out of execution with its registers restored.

Chapter *ASTs* contains further information about AST delivery.

19. SCH$INTERRUPT restores the scratch integer registers from the canonical kernel stack.
20. It executes a CALL_PAL REI instruction to pass control to the scheduled kernel thread through the interrupt/exception stack frame that formed the first part of the canonical kernel stack. Execution of the CALL_PAL REI instruction has the following additional effects:
    - The IPL is dropped from IPL$_SCHED.
    - The access mode is typically changed from kernel to a less privileged mode.

### 2.7.2 SWPCTX PALcode Routine

The swap process context routine, which executes in response to the CALL_PAL SWPCTX instruction, performs several operations. It expects to be entered with the following conditions:

- The PCBB processor register contains the physical address of the HWPCB for the current kernel thread.
- R16 contains the physical address of the HWPCB for the kernel thread that is to become current.

The PALcode routine performs the following operations:

1. It checks that current access mode is kernel, generating a privileged instruction exception if not.
2. It checks that the address in R16 is aligned on a 128-byte boundary.
3. It stores SP in the HWPCB for the current kernel thread as the value of the kernel stack pointer. It stores the per kernel thread stack pointers for the other three access mode stacks in the HWPCB unless this is a processor type that implements only the HWPCB forms of them.
4. It stores the ASTEN and ASTSR registers in the HWPCB unless this is a processor type that implements only the HWPCB forms of them.
5. It uses the process cycle counter to determine the current CPU cycles accumulated by the kernel thread and stores the longword value in the HWPCB.
6. It copies the process unique value from its internal storage unless this is a processor type that implements only the HWPCB copy.
7. At this point, all the privileged context of the current kernel thread has been saved, and the PALcode routine begins loading the new kernel thread's context.

First, it copies the contents of R16, the physical address of the HWPCB, to the PCBB processor register.

8. If this is a CPU type that does not support ASNs, it invalidates process-private TB entries.
9. It loads the set of processor registers whose contents it saved in the old kernel thread's HWPCB from the new kernel thread's HWPCB.

   In addition, it loads the PTBR, FEN, and DATFX registers, and the process unique value.
10. It calculates and loads the current value of this kernel thread's cycle counter.

The `CALL_PAL SWPCTX` instruction occurs in just a few locations in the executive:

- The rescheduling interrupt service routine executes this instruction to swap the context of the current kernel thread with that of the next kernel thread to be run. The current kernel thread might be still computable or being placed in a wait state.
- At the end of kernel thread deletion, the context of the kernel thread being deleted is switched for that of the system HWPCB with a `CALL_PAL SWPCTX` instruction. Kernel thread deletion completes in system HWPCB context.
- SMP$START_SECONDARY, executing on a secondary processor, swaps context from the termination HWPCB to the system HWPCB (see Chapter *Bootstrap Processing*).
- The swapper context switches to an alternative HWPCB so that it can adopt the page table space of a process being inswapped or outswapped.
- During a system crash, fatal bugcheck processing code adopts the page table space of a process whose address space is being written in a selective dump.
- During process creation, SWP$SHELL_INIT, in module SHELLxK, switches to an alternative HWPCB from the swapper context so that it can temporarily adopt the page table space of a process being created.

### 2.7.3 The Idle Loop

If SCH$INTERRUPT finds no computable kernel thread to execute, it swaps to the system HWPCB and executes code known as the idle loop. The idle loop's responsibilities are as follows:

- To shut down the CPU or enter the benign state if the CPU no longer has the run capability
- To execute a computable resident kernel thread as soon as one becomes available
- To maintain a supply of zeroed free pages, up to the maximum specified by SYSGEN parameter ZERO_LIST_HI, whose default value is 16

- To increment CPU$Q_IDLE_LOOP_COUNT as a count of iterations through the idle loop

The idle loop, routine SCH$IDLE in module SCHED_ROUTINES, performs the following operations:

1. It sets the bit corresponding to the CPU in SCH$GL_IDLE_CPUS to indicate that the CPU is idle.
2. It stores the address of the null PCB and a priority value of −1 in the CPU's per-CPU database.
3. Having made those changes, it can release the SCHED spinlock, lowering IPL to IPL$_RESCHED (3), the IPL of the rescheduling interrupt. This IPL permits software interrupts on this processor that can alter the scheduler database.
4. SCH$IDLE tests whether the CPU still has run capability and, if not, calls SMP$SHUTDOWN_CPU, described in Chapter *Symmetric Multiprocessing*.
5. It increments CPU$Q_IDLE_LOOP_COUNT.
6. If the supply of zeroed free pages needs replenishing, SCH$IDLE allocates a free page of physical memory whose virtual contents are no longer of interest and initializes the system PTE whose address is in CPU$L_ZEROED_PAGE_SPTE to map the allocated page. It zeros the page 32 bytes at a time, checking whether its bit in SCH$GL_IDLE_CPUS has been cleared in between chunks.

   The bit is cleared as a signal that a resident computable kernel thread is available. If the bit has been cleared, SCH$IDLE continues with step 9.

   The time during which the routine loops is counted as null time, which MONITOR displays as "Idle Time."

   After clearing an entire page, SCH$IDLE reinitializes its page frame number database record, increments the count of zeroed pages at MMG$GQ_ZEROED_LIST_COUNT, and inserts the page into the zeroed page list.
7. If the CLASS_SCHEDULER executive image is loaded (nonzero contents of SCH$GL_TBSH), SCH$IDLE calls SCH$CLASS_IDLE, in module SYSSCHED, described in Section 2.8.
8. It tests its idle bit. If the bit is set, it continues with step 4.
9. When this CPU's idle bit is cleared, SCH$IDLE sets bit CPU$V_SCHED in its per-CPU database to indicate that it is still idle and trying to acquire the SCHED spinlock. If an interval timer interrupt occurs while this bit is set, the interval timer interrupt service routine accounts for the CPU time as null time rather than as busy wait (MPSYNCH) time. SCH$IDLE tries to acquire the SCHED spinlock. Once successful, it clears CPU$V_SCHED and branches back to SCH$INTERRUPT to repeat the attempt to select a kernel thread to execute.

If another idle processor has already scheduled the computable kernel thread, this CPU may reexecute the idle loop.

## 2.8 CLASS SCHEDULING

OpenVMS optionally includes support for user-written class schedulers. The support is implemented in the optionally loadable executive image CLASS_SCHEDULER. This executive image consists of the Affect Process Scheduling ($SCHED) system service, by which a class scheduler application not supplied by Digital establishes a class scheduler process and routines to implement the underlying support for class scheduling.

The $SCHED system service enables an application to define different scheduling classes, each with a quantum value for the class as a whole, and to specify the class of each process the application can affect. Having established scheduling classes, the application can request the $SCHED system service to get information about each class and each process being class scheduled. A process is the unit associated with a class; all its kernel threads are subject to the time constraints of that class.

Periodically, the application requests the $SCHED system service to specify a new allotment of ticks. If the application fails to specify a new allotment within a given time period (default 30 seconds), the executive transforms all class-scheduled processes and kernel threads into normally scheduled processes and kernel threads.

Class scheduling support involves several additions to process data structures:

- The following bits in PCB$L_STS2, which by default are all clear in a newly created process:
  - Bit PCB$V_CLASS_SCHEDULED, when set, means that this process is subject to class scheduling.
  - Bit PCB$V_CLASS_SUPPLIED, when set, means that a class scheduler has specified a class for this process.
  - Bit PCB$V_WINDFALL, when set, means that the process is eligible for scheduling, even though its class is out of quantum, if there are no other eligible kernel threads to schedule.
- Bit KTB$V_IN_TBS_STATE in KTB$L_STS2, when set, means that the process's class is out of quantum and the kernel thread is not eligible for scheduling even though computable.
- PCB$L_CLASS_QUANT is the address of a class quantum data structure created by the $SCHED system service in nonpaged pool when a new class is defined.
- PCB$W_CLASS_EXTRA_TICKS is the number of extra ticks given to this process.

When the CLASS_SCHEDULER executive image is loaded, it initializes several system data cells. In particular, it initializes SCH$GL_TBS, SCH$GL_TBSW, and SCH$GL_TBSO with the addresses of its three arrays of KTB queues. Each array has 64 listheads, one for each internal scheduling priority. Effectively these are extensions to the COM and COMO queues. A kernel thread in one of these queues is said to be in a to be scheduled (TBS) state, although there is no corresponding SCH$C_*state* symbol for this state.

Nonzero contents in SCH$GL_TBS imply that the CLASS_SCHEDULER executive image has been loaded.

When a kernel thread reaches quantum end, SCH$QEND calls SCH$CLASS_QEND, in module SYSSCHED, to notify it of the quantum-end event. SCH$CLASS_QEND takes the following steps:

1. If the process is not class-scheduled, SCH$CLASS_QEND simply returns.
2. Otherwise, it checks whether the class quantum has enough ticks left to cover the replacement quantum just given to the kernel thread.
3. If there are some left but not a full quantum, SCH$CLASS_QEND reduces the quantum accordingly. It charges the quantum given against the class quantum and returns.
4. If the class is out of quantum, SCH$CLASS_QEND gives the kernel thread one tick. If no executive or kernel mode ASTs are pending for the kernel thread, SCH$CLASS_QEND sets bit KTB$V_IN_TBS_STATE. Otherwise, it leaves the bit clear so as not to block inner access mode execution in the kernel thread.
5. It returns to SCH$QEND.

If SCH$CLASS_QEND has set KTB$V_IN_TBS_STATE, SCH$QEND sets bit KTB$V_QUANTUM_RESCHED in KTB$L_STS2 and requests a rescheduling interrupt.

When the rescheduling interrupt is granted, if KTB$V_IN_TBS_STATE is set, the KTB is put, not on a COM queue, but on the appropriate queue for its priority: if the kernel thread's process is allowed windfall ticks, the KTB is inserted on a SCH$GL_TBSW queue; otherwise, it is inserted on a SCH$GL_TBS queue.

When a class-scheduled process is outswapped that has KTBs in either of those queues, each KTB is moved to the corresponding SCH$GL_TSBO queue.

When a favorable scheduling event is reported for a kernel thread that is in a TBS state, SCH$CHSEP checks the state of the kernel thread's executive mode, kernel mode, and special kernel mode AST queues. If the kernel thread is part of a multithreaded process and this kernel thread owns the inner mode semaphore, SCH$CHSEP also checks the processwide queues (see Chapter 5). If any of the queues is not empty, SCH$CHSEP clears KTB$V_

IN_TBS_STATE and changes the kernel thread's state to COM or COMO, thus allowing the inner mode AST to execute even though the scheduling class is out of quantum.

When there are no COM kernel threads to schedule on a CPU, rather than leave it idle the executive will schedule any kernel thread whose class is out of quantum but that is eligible for windfall ticks. The idle loop, described in Section 2.7.3, calls SCH$CLASS_IDLE, in module SYSSCHED. SCH$CLASS_IDLE scans the TBSW queues, starting with the highest priority queue, looking for a nonempty queue. If it finds one, it acquires the SCHED spinlock and reexamines the queue. If there is indeed a KTB, SCH$CLASS_IDLE clears the KTB$V_IN_TSB_STATE bit, gives the kernel thread a quantum of five ticks, and calls SCH$CHSE to change the kernel thread's state to COM. SCH$CLASS_IDLE releases the SCHED spinlock and returns to SCH$IDLE, which will attempt to schedule the now computable kernel thread.

When SCH$PIXSCAN (see Section 2.6.7) runs, if the CLASS_SCHEDULER executive image has been loaded, SCH$PIXSCAN calls SCH$CLASS_PIXSCAN, in module SYSSCHED. SCH$CLASS_PIXSCAN scans the CLASS_SCHEDULER image's TBS and TBSW queues, starting with the highest priority queue. It potentially changes the state of PIXSCAN number of kernel threads from TBS to COM. It gives each kernel thread a quantum of three ticks and clears its KTB$V_IN_TBS_STATE bit.

## 2.9 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIB]CPUDEF.SDL
[LIB]FREDDEF.SDL
[LIB]PCBDEF.SDL
[LIB]PHDDEF.SDL
[LIB]RSNDEF.SDL
[LIB]STATEDEF.SDL
[LIB]WQHDEF.SDL
[SYS]CAPABILITY.MAR
[SYS]MUTEX.MAR
[SYS]RSE.MAR
[SYS]SCHED_ROUTINES.MAR
[SYS]SCHEDULER.M64
[SYS]SYS_AFFINITY.C
[SYS]SYSSCHED.B32
[SYS]SYSWAIT.MAR

# 3 Event Flags

I claim not to have controlled events, but confess plainly that events have controlled me.

Abraham Lincoln, Letter to A. G. Hodges, April 4, 1864

Event flags are numbered status bits maintained by the OpenVMS Alpha operating system for general programming use. Each event flag can be either set or clear, and its status can be tested.

System services read, set, and clear event flags. A program can specify that an event flag be set at the completion of an operation such as an I/O request. When the program can proceed no further until the request is complete, it can request a system service to wait for the event flag to be set.

This chapter describes the implementation of event flags and the services that support them.

## 3.1 EVENT FLAGS

An event flag can be used for synchronization with the completion of certain system services, such as I/O, lock, informational, and timer requests. Each of these services includes an argument identifying the number of the event flag to be associated with the request. When a program requests such a system service, that event flag is cleared. It is subsequently set when the request has been completed as a signal that the operation is complete. Event flags can also be used as application-specific synchronization tools.

Event flags can be local to one process or shared among processes in the same user identification code (UIC) group. Each process has 64 local (process-specific) event flags, in two clusters of 32 flags each. Local event flags are processwide resources and can be used by all the threads of a process.

Event flags shared among processes are called common event flags. In addition to local event flags, a process can access up to 64 common event flags at once, in two clusters of 32 flags each. Before a process can refer to the flags in a particular common event flag cluster, it must explicitly associate with the cluster (see Section 3.1.3). The common event flags in a cluster can be used by all the threads of each process that has associated to the cluster. Processes sharing common event flags must be running on a single VMScluster member; that is, common event flags are not visible clusterwide.

The operating system assigns no inherent meaning to any particular event flag, although it does reserve certain flags for its own use (see Section 3.1.1). A program defines the meaning of a flag by the way it uses the flag.

For example, when a program requests the Queue I/O Request ($QIO) system service, specifying event flag 10 as the EFN argument, it can subsequently wait for completion of that I/O request by waiting for event flag 10 to be set. When the I/O completes, the executive sets event flag 10 as a signal to end the wait. After the wait is satisfied, the meaning of event flag 10 is undefined.

With multithreading support enabled in OpenVMS Alpha Version 7.0, a program runs in the execution context of a kernel thread within a process. If a process has only its initial kernel thread and no user mode thread manager, when the program waits for an event flag the process's only kernel thread is placed into a wait, and program execution is stalled until the flag is set and the kernel thread placed back into execution.

If a program has been built with the DECthreads run-time library, it can create many more user mode threads than it has kernel threads. Each kernel thread (or the only kernel thread) is an execution context for multiple user mode threads. When a user mode thread waits for an event flag, the DECthreads thread manager is notified so that it can schedule a different user mode thread in the kernel thread.

Some of the system services for which an event flag argument can be specified are Breakthrough [and Wait] ($BRKTHRU[W]), Enqueue Lock Request [and Wait] ($ENQ[W]), and Queue I/O Request [and Wait] ($QIO[W]).

A program can request the asynchronous form of such a service, for example, $QIO; do whatever work it can without the I/O request's having completed; and then wait for completion of the I/O request by requesting a wait service such as Wait for Single Event Flag ($WAITFR) or Synchronize ($SYNCH). Alternatively, a program can request the synchronous form of a system service, in this example, $QIOW, which effectively consists of a $QIO request followed by a $SYNCH request.

The $SYNCH system service (see Section 3.6.3) tests for completion of a previously requested operation by examining the status block associated with the operation; the event flag associated with the operation merely acts as a trigger to make the process computable after a wait so that the status block can be tested.

Even a single-threaded process can have multiple threads of execution: mainline code and asynchronous system trap (AST) procedures that interrupt it. The mainline code and its AST procedures may request asynchronous operations. Moreover, system services requested by any of these threads may request asynchronous operations. If multiple concurrent asynchronous operations within a process use the same event flag, the meaning of its being set or clear is ambiguous. Under some circumstances, concurrent uses of the same flag within a process can lead to a spurious signal of completion; under others, concurrent uses can lead to an indefinite wait. Section 3.9 explains the potential problems and discusses ways to avoid them.

### 3.1.1 Local Event Flags

The 64 local event flags are contained in each process's process header (PHD), at offset PHD$Q_LEFC. The quadword is divided into two longwords whose names are PHD$L_LEFC_0 and PHD$L_LEFC_1. All local event flags are initialized to zero during process creation.

OpenVMS VAX local event flags are in a process's process control block (PCB). Their relocation in OpenVMS Alpha to the PHD makes it possible for them to be read directly by outer mode code, as described in Section 3.6.1.

Local event flags 0 to 31 make up cluster 0. Bit 0 in PHD$L_LEFC_0 corresponds to event flag 0, bit 1 to event flag 1, and so on. Local event flags 32 to 63 make up cluster 1. Bit 0 in PHD$L_LEFC_1 corresponds to event flag 32, bit 1 to event flag 33, and so on.

Event flag 0 is the default event flag supplied by certain compilers whenever a program requests a system service that takes an event flag number argument, but does not specify a particular flag. Consequently, it is more likely than others to be used incorrectly for multiple concurrent requests.

Event flag numbers 24 through 31 are reserved for system use; this means they can be set and cleared at any time by executive software and should not be used by application software.

Local event flags are set or cleared using the load-locked/store-conditional mechanism described in Chapter *Synchronization Techniques*. This is sufficient to ensure that concurrent or nearly concurrent writes to the longword containing a cluster of flags all take effect. Code that tests or sets an event flag must first have acquired the SCHED spinlock to serialize access to the scheduler database in case a kernel thread's state must be changed. Although the spinlock is not necessary to synchronize access to the flags, it must be acquired first so as to make atomic the act of testing or setting the flag and the act of changing the scheduler database.

### 3.1.2 Event, No Flag

Most OpenVMS system services that incorporate an asynchronous operation have an event flag argument. Each such service must clear the flag before the operation is initiated, and the flag must be set when the operation completes. To reduce overhead in cases where the value of the flag is irrelevant to the program, OpenVMS Alpha Version 7.0 adds support for an event with no associated flag. The symbol corresponding to the "flag number" of this event is EFN$C_ENF; its numeric value is 128. When EFN$C_ENF is passed to a service that incorporates an asynchronous operation, the service does not clear a flag at the start of the operation, and a flag is not set at the end of the operation.

EFN$C_ENF is always interpreted as being set; it cannot be cleared. When it is passed to a service that reads, clears, or sets event flags, the service

simply returns the status SS$_WASSET. Because EFN$C_ENF is always interpreted as set, when it is used as an argument to the $WAITFR service, the service never waits; it returns immediately with the status SS$_WASSET. EFN$C_ENF is not a legal argument to either the Wait for Logical OR of Event Flags ($WFLOR) or the Wait for Logical AND of Event Flags ($WFLAND) service.

EFN$C_ENF is primarily intended as an argument to the $SYNCH system service, whether requested explicitly or implicitly through a system service such as $QIOW. EFN$C_ENF is also used implicitly by the Perform Fast I/O ($IO_PERFORM) system service.

Strictly speaking, EFN$C_ENF is not a flag and thus is neither local nor common. However, a kernel thread passing that flag to the $SYNCH service is placed into a local event flag wait if the status is still zero (see Section 3.6.3). When an operation completes for which EFN$C_ENF has been specified, the executive makes computable all kernel threads of the associated process that are in local event flag wait for EFN$C_ENF (see Section 3.7).

### 3.1.3 Common Event Flags

A program creates a common event flag cluster dynamically, by requesting the Associate Common Event Flag Cluster ($ASCEFC) system service (see Section 3.3). Each common event flag cluster is described by a nonpaged pool data structure called a common event block (CEB), whose layout is shown in Figure 3.1.

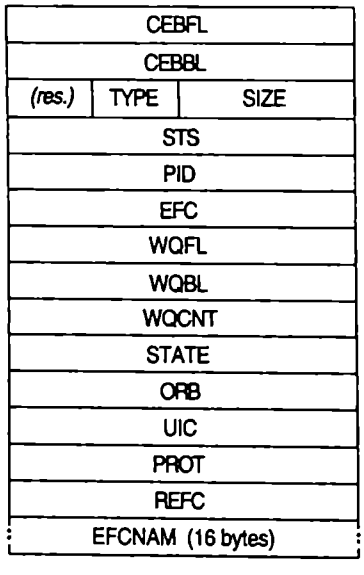



**Figure 3.1**
Layout of Common Event Block (CEB)

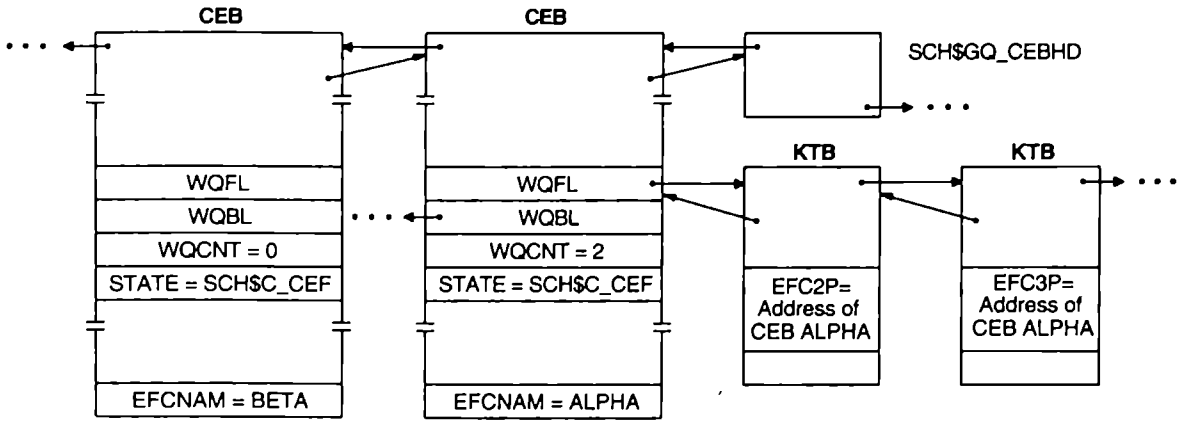



**Figure 3.2**
Common Event Flag Wait Queues

The program specifies whether it will access the flags in that cluster using event flag numbers 64 through 95 (cluster 2) or 96 through 127 (cluster 3). No other cluster number or set of flags can be used to specify common event flags. If the flags are associated as cluster 2, the field PCB$L_EFC2P contains the address of their CEB. Otherwise, PCB$L_EFC3P contains its address.

CEB$L_CEBFL and CEB$L_CEBBL link each CEB into a systemwide list whose listhead is SCH$GQ_CEBHD (see Figure 3.2). The system global SCH$GW_CEBCNT contains the number of CEBs in the list. The mutex EXE$GL_CEBMTX synchronizes access to the CEB list. Chapter *Synchronization Techniques* describes the use of mutexes.

A particular common event flag cluster is identified by its name, CEB$T_EFCNAM, and UIC group, CEB$W_GRP. (CEB$W_GRP is the high-order word of CEB$L_UIC.) There cannot be more than one cluster with the same name and group. Consequently, only processes within the same UIC group can share common event flags. This minimizes the likelihood of conflicting names within the system.

Once a common event flag cluster exists, other processes in the same UIC group can associate to it. Once any kernel thread in a process has requested the $ASCEFC service to associate to a common event flag cluster, all kernel threads in that process can access the cluster's flags.

Two bits are defined in CEB$L_STS:

- CEB$V_PERM, when set, indicates that the cluster is a permanent one rather than a temporary one.
- CEB$V_NOQUOTA, when set, indicates that no quota was charged for the creation of the cluster.

Creation of a temporary cluster is charged against a job's timer queue entry (TQE) quota. Creation of a permanent cluster uses no quota but requires the

privilege PRMCEB. A temporary cluster exists only as long as a process is associated to it, whereas a permanent cluster must be explicitly deleted (see Section 3.5).

CEB$L_PID contains the internal process ID (IPID) of the master process in the job tree of the process that created the cluster.

The field CEB$L_EFC contains the 32 event flags. These are all initialized to zero when the cluster is created.

The fields CEB$L_WQFL and CEB$L_WQBL, CEB$L_WQCNT, and CEB$L_STATE form a wait queue (see Chapter 2) for kernel threads waiting for flags in that cluster. CEB$L_STATE contains the constant SCH$C_CEF, and CEB$L_WQCNT the number of kernel threads waiting for flags in that cluster. The queue can contain kernel threads from any of the processes that have associated to the cluster.

CEB$L_UIC contains the UIC of the creating process. CEB$L_PROT contains the value of the PROT argument on the $ASCEFC request that created the cluster. This field is no longer used in checking access to the cluster. Instead, the object rights block (ORB) associated with the cluster contains the security profile of the cluster, defining its object security characteristics, for example, owner, protection mask, and access control list entries. The security profile is initially based on the creator's UIC and the PROT argument but may be subsequently modified, for example, through the EDIT/ACL utility.

CEB$L_REFC contains the number of processes that are currently associated to the cluster.

Like local event flags, common event flags are set and cleared with the load-locked/store-conditional mechanism. The same synchronization considerations apply to both types of flag.

## 3.2 PCB AND KTB FIELDS RELATED TO EVENT FLAGS

Figure 3.3 shows the PCB and KTB fields related to the use of event flags. The KTB fields are shaded. The KTB for a single-threaded process with only the initial kernel thread overlays its PCB. A multithreaded process has an additional KTB for each additional kernel thread.

When a process is outswapped, its PHD may be outswapped as well, leaving the local event flags in its PHD inaccessible to a routine trying to set any of them (see Section 3.7). When outswapping a process, the swapper therefore copies the local event flags from the PHD to PCB$Q_LEFC_SWAPPED. The quadword is divided into two longwords whose names are PCB$L_LEFC_0_SWAPPED and PCB$L_LEFC_1_SWAPPED. When inswapping a process, the swapper copies the flags from the PCB to the PHD.

The local event flags in the PHD and the copy in the PCB are process-wide resources. In addition, associations to common event clusters affect the

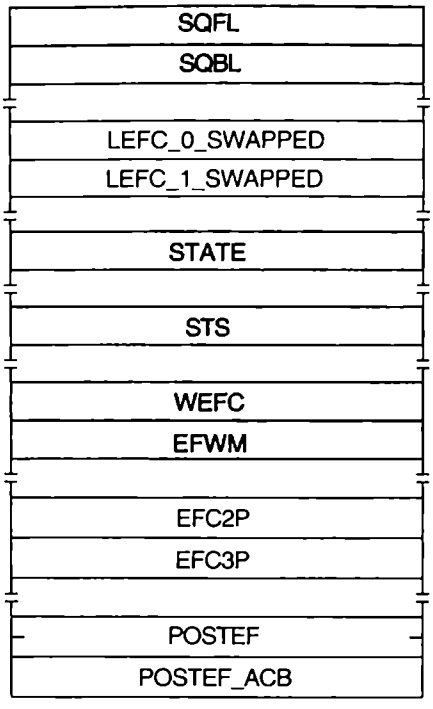



**Figure 3.3**
Process Control Block (PCB) and Kernel Thread Block (KTB) Fields That Support Event Flags

process as a whole; thus the fields that store addresses of associated common event flag clusters are in PCB$L_EFC2P and PCB$L_EFC3P. Section 3.1.3 contains further information on these two fields.

PCB$Q_POSTEF and PCB$L_POSTEF_ACB are used in reporting the setting of a local event flag to a user mode thread manager. Chapter 5 discusses user mode thread manager upcalls.

The shaded fields in Figure 3.3 are significant for a kernel thread in an event flag wait and thus are in the KTB. KTB$L_WEFC contains the number of the cluster containing the flags for which a kernel thread waits. For a kernel thread waiting for EFN$C_ENF, KTB$L_WEFC contains the value 4. KTB$L_EFWM contains a mask that is the 1's complement of the flags in the cluster for which the kernel thread is waiting. The KTB$L_STS bit KTB$V_WALL, when set, indicates that the kernel thread is waiting for all those flags to be set.

The KTB event flag fields are loaded only when a kernel thread initiates an event flag wait. Consequently, for a kernel thread in a state other than event flag wait, they may be stale. Furthermore, the field KTB$L_EFWM has an additional use: it identifies the resource waited for by a kernel thread in a miscellaneous wait state (see Chapter 2).

## 3.3 ASSOCIATING TO A COMMON EVENT FLAG CLUSTER

A program requests the $ASCEFC system service to create a named common event flag cluster if it does not already exist and to access its flags. The program specifies the name of the cluster and implicitly, through the process's PCB$L_UIC field, the UIC group of the cluster.

The $ASCEFC system service procedure, EXE$ASCEFC in module SYSASCEFC, runs in kernel mode. It takes the following steps:

1. EXE$ASCEFC confirms that the event flag number is within cluster 2 or 3, returning the error status SS$_ILLEFC if it is not.
2. It checks that the cluster name string specified by the NAME argument is accessible, returning SS$_ACCVIO if it is not.
3. It locks the CEB mutex for write access.
4. It searches the CEB list for a cluster with the specified cluster name and the same UIC group as the requesting process.
5. If one exists, EXE$ASCEFC checks whether the process can access it, based on the protection and access specified in the ORB. The access required is called associate access. Consult the *OpenVMS Guide to System Security* for further information on access checking.

   If the process is allowed access, EXE$ASCEFC continues with step 8. Otherwise, it unlocks the mutex and returns the error status SS$_NOPRIV.
6. If the common event flag cluster does not already exist, the process is requesting its creation. EXE$ASCEFC makes several checks:
   - The cluster name string must comprise 15 or fewer characters. If it does not, the routine unlocks the mutex and returns the error SS$_IVLOGNAM.
   - If the process requests a permanent cluster, it must have the privilege PRMCEB. If it does not have the privilege, EXE$ASCEFC unlocks the mutex and returns the error status SS$_NOPRIV.
   - If the process is not requesting a permanent cluster, EXE$ASCEFC charges it by decrementing the job's TQE quota. If the job has insufficient quota, EXE$ASCEFC unlocks the mutex and returns the error status SS$_EXQUOTA.
7. EXE$ASCEFC allocates a CEB from nonpaged pool and initializes it. It sets CEB$V_PERM in CEB$L_STS if the cluster is a permanent one. It increments SCH$GW_CEBCNT, the number of CEBs, and links the new CEB into the list.
8. Whether or not the cluster existed previously, the routine associates the process and the cluster by incrementing the cluster's reference count and by storing the address of the CEB in either PCB$L_EFC2P or PCB$L_EFC3P.

   First, however, it saves the old contents of PCB$L_EFC2P or PCB$L_EFC3P. If they are not zero, the process has been using the correspond-

ing event flag numbers to associate with another cluster. If the old contents and address of the newly associated CEB are identical, the process is reassociating to a cluster with which it was previously associated. EXE$ASCEFC decrements the cluster's reference count to balance the increment just done, unlocks the mutex, and returns.

If the two addresses are different, EXE$ASCEFC severs the connection between the process and the previously associated cluster by taking steps 5, 7, and 8 in Section 3.4.

9. It unlocks the mutex and returns.

## 3.4 DISSOCIATING FROM A COMMON EVENT FLAG CLUSTER

A program dissociates its process from a common event flag cluster explicitly by requesting the Disassociate Common Event Flag Cluster ($DACEFC) system service with an event flag number within that cluster. Implicitly, the service is requested on behalf of the process when a program running in it associates a new event flag cluster with a cluster number already in use.

The $DACEFC system service procedure, EXE$DACEFC in module SYSASCEFC, runs in kernel mode. It takes the following steps:

1. EXE$DACEFC confirms that the event flag number is within cluster 2 or 3, returning the error status SS$_ILLEFC if it is not.
2. It locks the CEB mutex for write.
3. It locates the CEB using the PCB's pointer to the cluster, either PCB$L_EFC2P or PCB$L_EFC3P.
4. If the pointer was zero, the process did not have an associated cluster. The routine simply unlocks the mutex and returns.
5. Otherwise, if the process does have an associated cluster, EXE$DACEFC tests whether the process has more than one kernel thread. If so, it acquires the SCHED spinlock and scans the queue of KTBs waiting for flags in this cluster.

   For each queued KTB that represents a kernel thread in this process, EXE$DACEFC reports an event-flag-setting event (see Section 3.7) for the kernel thread by calling SCH$REPORT_EVENT, in module RSE. As a result, the KTB is removed from the CEB wait queue, and the kernel thread is made computable. When it is subsequently placed into execution, it will repeat its event flag wait test and find that there is no longer an associated cluster. The wait system service will return SS$_ILLEFC to that kernel thread.
6. EXE$DACEFC clears either PCB$L_EFC2P or PCB$L_EFC3P.
7. It decrements the cluster's reference count. If there are other processes associated to the cluster or if the cluster is a permanent one, it unlocks the mutex and returns.
8. Otherwise (if the cluster is temporary and has no processes still associated with it), EXE$DACEFC deletes it by taking the following steps:

a. If CEB$V_NOQUOTA in CEB$L_STS is clear, EXE$DACEFC increments the TQE quota of the job whose quota was originally charged.
b. EXE$DACEFC decrements SCH$GW_CEBCNT, removes the CEB from the CEB list, deallocates it to nonpaged pool, and deallocates its associated ORB to paged pool.
c. EXE$DACEFC unlocks the mutex and returns.

During image rundown, a process is automatically dissociated from any common event flag clusters to which it had associated.

## 3.5 DELETING A COMMON EVENT FLAG CLUSTER

To delete a permanent event flag cluster, a program requests the Delete Common Event Flag Cluster ($DLCEFC) system service with the name of the cluster to be deleted.

A cluster cannot be deleted if processes are still associated with it. In such a case, the $DLCEFC service transforms the permanent cluster to a temporary one so that it will be deleted when the last process associated with the cluster requests the $DACEFC service.

The $DLCEFC system service procedure, EXE$DLCEFC in module SYSASCEFC, runs in kernel mode. It takes the following steps:

1. It checks that the cluster name string specified by the NAME argument is accessible, returning SS$_ACCVIO if it is not.
2. EXE$DLCEFC locks the CEB mutex for write.
3. It scans the CEB list for a cluster of the specified name and a group code matching that of the process (PCB$W_GRP). If it fails to find one, it unlocks the mutex and simply returns SS$_NORMAL as the service status.
4. If it finds one, it tests whether the process is allowed to delete the cluster based on the protection and access specified in the ORB. If the process is not allowed to delete the cluster, EXE$DLCEFC unlocks the mutex and returns the error status SS$_NOPRIV.
5. Unless the process is deleting a temporary CEB, EXE$DLCEFC clears CEB$V_PERM and sets CEB$V_NOQUOTA in CEB$L_STS. This effectively changes the cluster to a temporary one for which no quota need be returned. The cluster's deletion is always deferred until all processes have dissociated from it.
6. EXE$DLCEFC increments the cluster's reference count and transfers to code within EXE$DACEFC, described in step 7 in Section 3.4. (The increment balances a decrement in EXE$DACEFC.)

## 3.6 WAITING FOR AN EVENT FLAG

A kernel thread can be placed into an event flag wait state to wait for the setting of one or more flags. When a kernel thread waits for more than one

flag, all the flags must be in the same cluster. A kernel thread waits for event flags by performing any of the following actions:

- Requesting one of the three event flag wait system services directly: SWAITFR, SWFLOR, or SWFLAND
- Requesting the $SYNCH system service, which combines an event flag wait and a status block test to wait for service completion, thus minimizing spurious completions caused by multiple concurrent uses of the same flag (see Section 3.9)
- Requesting the wait form of services, for example, $QIOW or $ENQW, each of which incorporates $SYNCH
- Requesting synchronous Record Management Services (RMS) operations, which results in RMS's requesting $SYNCH

Synchronous RMS services are described in Chapter *System Service Dispatching*. The wait and $SYNCH services are described in the sections that follow.

### 3.6.1 Mode of Caller Event Flag Wait Services

Only the single flag specified on a $WAITFR request must be set for the wait condition to be satisfied. The other two services involve multiple flags.

The distinction between $WFLOR and $WFLAND lies in how many of the flags must be set for the wait condition to be satisfied. One or more of the flags in the mask must be set to satisfy the wait condition for $WFLOR. Each of the flags specified in the $WFLAND system service argument must have been set for the wait to be satisfied. Although the flags need not be set simultaneously, clearing one of the flags in the mask before all are set can result in an indefinite wait, as described in Section 3.9.

The wait system service procedures, EXE$WAITFR, EXE$WFLOR, and EXE$WFLAND, which are in module SYSWAIT, run in the mode of the service requestor. The three service procedures have the same form. Each checks whether the flag or flags are local or common and whether the wait condition is already met:

- If the flags are local, each service tests the flags in the PHD to see if the wait condition is already met and, if so, executes a memory barrier (MB) instruction and returns the status SS$_WASSET.

  An event flag can be set asynchronously by a thread of execution running on another processor as a signal that some process state can now be accessed, for example, the contents of an I/O buffer. The explicit memory barrier orders reading the flag with reading the other process state. Chapter *Synchronization Techniques* discusses memory barriers and the need to order memory accesses in multiprocessor Alpha systems.

  When the wait condition is already met, the services can complete without the overhead of changing mode to kernel.

- If the flag is EFN$C_ENF (which is legal only with $WAITFR), the service executes an MB instruction and returns the status SS$_WASSET.
- If the flags are local, but the wait condition is not met, each service tests whether the process is multithreaded and, if so, whether it is appropriate to notify a user mode thread manager. If it is, the service calls EXE$INFORM_TM_CALL, in module TM_SUPPORT, to call the user mode thread manager. It may save the context of this user mode thread and switch to a different one. When the stalled user mode thread's event flag wait is satisfied, the user mode thread manager places it back into execution at a successful return from the wait service. Chapter 5 discusses user mode thread management.

  If the process is not multithreaded or it is not appropriate to notify a user mode thread manager, each service requests a corresponding system service whose name has the form $*wait*_INT, for example, $WAITFR_INT.
- If the flags are common, each service requests a corresponding system service whose name has the form $*wait*_COMMON, for example, $WFLOR_COMMON.

### 3.6.2 Kernel Mode Event Flag Wait Services

The system service procedures that correspond to the mode of caller event flag wait services are in module SYSWAIT. Each executes in kernel mode. Each EXE$*wait*_COMMON and EXE$*wait*_INT procedure copies the wait mask argument and transfers control to EXE$WAIT, in module SYSWAIT.

EXE$WAIT is entered with a mask identifying the flags to be waited for, the number of a flag in that cluster, and a wait-all flag that is set if the entry is from $WFLAND.

EXE$WAIT takes the following steps:

1. It checks that the event flag number is legal, returning the error status SS$_ILLEFC if the number is out of range.
2. EXE$WAIT raises interrupt priority level (IPL) to 2 to block delivery of a kernel mode AST procedure that might request another wait service.
3. It determines which cluster contains that event flag and records the cluster number in KTB$L_WEFC.
4. If the cluster number is 2 or 3, indicating a common event flag cluster, EXE$WAIT first checks that there is an associated cluster and returns the error status SS$_UNASEFC if there is none.

   If there is an associated cluster, it gets the CEB address from either PCB$L_EFC2P or PCB$L_EFC3P, depending on the cluster number.
5. It tests whether proactive memory reclamation from processes that wake periodically has been enabled and whether this process is potentially subject to working set reduction (see Chapter *Working Set List Dynamics*). If so, it acquires the MMG spinlock, raising IPL to IPL$_

MMG. EXE$CHK_WAIT_BHVR, potentially called in a later step, must be called with the MMG spinlock already acquired to avoid a spinlock rank violation (see Chapter *Synchronization Techniques*).

6. EXE$WAIT acquires the SCHED spinlock, raising IPL to IPL$_SCHED, to synchronize access to the scheduler database. (The spinlock acquisition results in implicit memory barriers.)
7. It tests whether the event flag wait condition is satisfied by the current state of the flags.
8. If the wait condition is satisfied, EXE$WAIT releases the SCHED spinlock and, if it had been locked, the MMG spinlock. It returns to the mode of caller service procedure, which had requested either $*wait*_INT or $*wait*_COMMON. The mode of caller service returns to its requestor.
9. If the event flag wait condition is unsatisfied and if this process is potentially subject to working set reduction, it calls EXE$CHK_WAIT_BHVR, in module RSE, to reduce the working set if appropriate. EXE$WAIT then releases the MMG spinlock. Chapter 2 describes the routine and this form of memory reclamation in more detail.
10. EXE$WAIT checks whether the wait is wait-all. If so, it sets the KTB$V_WALL bit in KTB$L_STS.
11. EXE$WAIT stores a mask representing the flags to be waited for in KTB$L_EFWM:
    - If the program requested $WFLOR, the mask is the 1's complement of the input mask passed to the system service.
    - If the program requested $WAITFR, the mask consists of a 1 in every bit except the bit number corresponding to the specified flag. (The $WAITFR mask is thus a special case of a wait for any one of a group of flags to be set.)
    - If the program requested $WFLAND, EXE$WAIT clears any bits in the input mask corresponding to currently set flags, complements it, and then stores it in KTB$L_EFWM.
12. EXE$WAIT calls SCH$WAIT_PROC, in module SCHED_ROUTINES, to change the kernel thread's scheduling state to either a local or a common event flag wait state, depending on the cluster number. SCH$WAIT_PROC returns the status SS$_WAIT_CALLERS_MODE to the change mode dispatcher. In response to that status the change mode dispatcher initiates a context switch. Chapter 2 describes this sequence in greater detail.

A crucial point in this implementation is that the kernel thread waits at the access mode associated with the original synchronous system service request, thus allowing AST delivery to all access modes at least as privileged as that of the synchronous service request. In the usual case, an AST of any access mode can be delivered while the kernel thread is waiting for the

service to complete. The PC (program counter) associated with the wait is within the mode of caller service. When the kernel thread is placed back into execution, the service repeats the event flag test.

Figure 3.4 shows an example flow through $WAITFR and $WAITFR_INT. The program first requests SQIO, specifying event flag 10. The $QIO service clears event flag 10, initiates the I/O request, and returns to the progam. Later, the program requests $WAITFR, also specifying event flag 10.

Initially, the flag is clear and there is no user mode thread manager to notify. EXE$WAITFR requests $WAITFR_INT, which transfers to EXE$WAIT. Holding the SCHED spinlock, EXE$WAIT tests the state of the flag and, finding it clear, places the kernel thread into a local event flag (LEF) wait.

At some later time, the previously requested I/O request completes. The special kernel mode AST associated with the completion of that request is queued to the waiting kernel thread, causing it to become computable. When the thread is placed back into execution, it executes the special kernel mode AST code, which sets local event flag 10. When the AST returns, the program resumes its execution of EXE$WAITFR. EXE$WAITFR repeats its test to determine whether the flag is set. This time the flag is set, and EXE$WAITFR returns to its requestor.

### 3.6.3 The $SYNCH System Service

The $SYNCH system service procedure, EXE$SYNCH in module SYSSYNCH, executes in the mode of its caller. It is typically requested with the number of an event flag and the address of a status block or an I/O status area. It simply branches to the routine EXE_STD$SYNCH_LOOP, the same routine called by synchronous system services such as $QIOW and $ENQW. (Some of the services call EXE$SYNCH_LOOP, in SYSSYNCH, a jacket routine for EXE_STD$SYNCH_LOOP.)

EXE_STD$SYNCH_LOOP takes the following steps to determine whether the requested operation is complete:

1. It first tests whether a status block was specified by the requestor. For $GETLKIW and $ENQW, the lock status block would serve this purpose; for $IO_PERFORM[W], the I/O status area would; and in all other cases, an I/O status block (IOSB) would be used.

   If no status block was specified, EXE_STD$SYNCH_LOOP requests the $WAITFR system service to place the kernel thread into an event flag wait state until the specified flag is set. When the $WAITFR service returns, EXE_STD$SYNCH_LOOP returns to its caller, either EXE$SYNCH or a synchronous system service.
2. If a status block was specified, EXE_STD$SYNCH_LOOP tests the status field in it. A nonzero status indicates that the asynchronous service has completed, and EXE_STD$SYNCH_LOOP returns to its caller. Note that if the status block is being used concurrently for multiple requests,

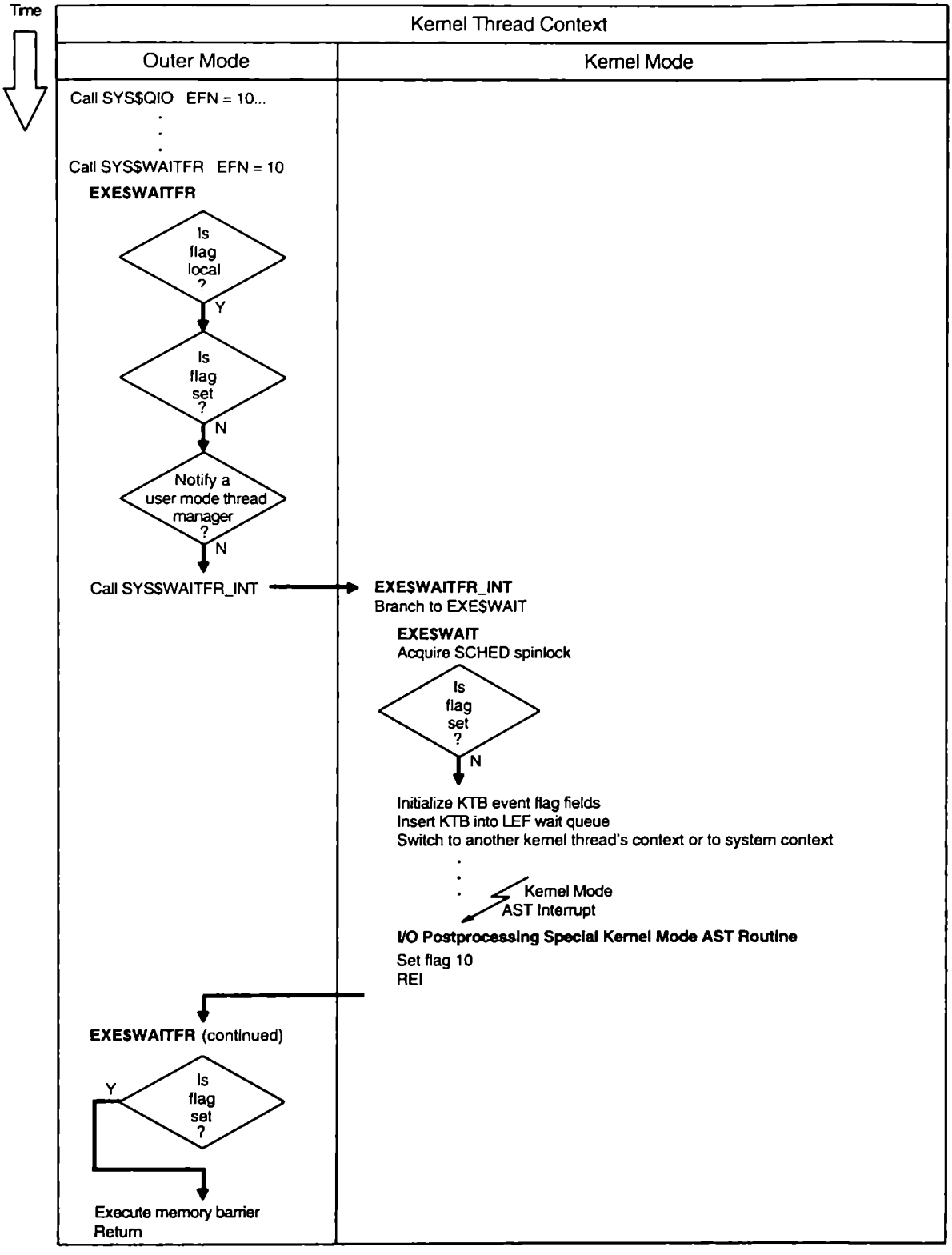



**Figure 3.4**
Example $WAITFR Flow

one or both of the requested services can complete prematurely; the $SYNCH service assumes that nonzero contents imply asynchronous service completion and does not test whether the event flag is set.

A zero status field indicates that the asynchronous service has not yet completed.

3. If the status is zero, EXE_STD$SYNCH_LOOP determines what kind of event flag was specified and, possibly, whether the process is multithreaded.

   - If the flag is local or EFN$C_ENF, it tests whether the process is multithreaded and, if so, whether it is appropriate to notify a user mode thread manager. If it is, EXE_STD$SYNCH_LOOP calls EXE$INFORM_TM_CALL, in module TM_SUPPORT, to call the user mode thread manager. The user mode thread manager may save the context of this user mode thread and switch to a different one. When the previous thread's event flag wait is satisfied, it will be placed back into execution, returning successfully from the $SYNCH service. Chapter 5 discusses user mode thread management.
   - If the flag is local but the process is not multithreaded or has no user mode thread manager that requested an event flag wait upcall, EXE_STD$SYNCH_LOOP requests the internal kernel mode form of the service, $SYNCH_INT, described in Section 3.6.4.

     If the status is still zero, $SYNCH_INT places the kernel thread into a local or common event flag wait. The wait PC is code within EXE_STD$SYNCH_LOOP, and the access mode at which the kernel thread is waited is the mode of the $SYNCH requestor.
   - If the flag is common, EXE_STD$SYNCH_LOOP requests $SYNCH_INT.

4. When the kernel thread is made computable again, possibly as the result of an event-flag-setting event (see Section 3.7), EXE_STD$SYNCH_LOOP tests the status field. If it is nonzero, EXE_STD$SYNCH_LOOP returns to its caller.
5. If the status field is zero, then the kernel thread has been made computable before the wait should be ended. An AST might have been queued to the kernel thread, or possibly the event flag was set spuriously, perhaps by another concurrent use. EXE_STD$SYNCH_LOOP rerequests $SYNCH_INT.

### 3.6.4 The $SYNCH_INT System Service

The $SYNCH_INT system service procedure, EXE$SYNCH_INT in module SYSSYNCH, executes in kernel mode, requested from EXE_STD$SYNCH_LOOP. Following is a simplified description of its actions to determine whether the requested operation is complete and, if not, to wait the kernel

thread. The description omits such details as checks for proactive memory reclamation and the possibility of a status block crossing a page boundary.

EXE$SYNCH_INT takes the following steps:

1. It raises IPL to 2.
2. It gets the address of the KTB for this kernel thread.
3. It determines what kind of event flag was specified, the address of the cluster containing it, and the address of the associated wait queue.
4. It stores the cluster number of this flag in KTB$L_WEFC preparatory to placing the kernel thread into a wait.
5. EXE$SYNCH_INT checks that the status block is accessible, returning SS$_ACCVIO if not.
6. It examines the status field.
7. If the status field is nonzero, it returns SS$_WASSET to its caller.
8. If the status field is zero, it stores the address of the status block in PCBSQ_KEEP_IN_WS to ensure that the page containing the status block remains in the process's working set, so that it can be accessed at an IPL above 2. It examines the status field again, possibly faulting the status block into the process's working set.
9. It acquires the SCHED spinlock so as to make atomic the sequence of testing the status word and waiting the kernel thread.
10. It tests the status field again. It is because of this test at IPL$_SCHED that the status block had to be faulted into the working set. If the status is nonzero, indicating that the operation has completed, EXE$SYNCH_INT releases SCHED, reinitializes PCB$Q_KEEP_IN_WS, and returns SS$_WASSET to its requestor.
11. If the status field is still zero, it reinitializes PQB$Q_KEEP_IN_WS. It clears the specified event flag unless the flag is EFN$C_ENF.
12. EXE$SYNCH_INT initializes the rest of the kernel thread's event flag wait fields: it clears KTB$V_WALL in KTB$L_STS and stores in KTB$L_EFWM a mask of all 1's except for the bit number of the specified flag. (KTB$L_WEFC was initialized in step 4.)
13. It calls SCH$WAIT_PROC, in module SCHED_ROUTINES, to change the kernel thread's scheduling state to either a local or a common event flag wait state, depending on the cluster number. EXE$SYNCH_INT returns the status SS$_WAIT_CALLERS_MODE to the change mode dispatcher. In response to that status the change mode dispatcher initiates a context switch.

The kernel thread waits at the access mode associated with the original synchronous system service request, thus allowing AST delivery to all access modes at least as privileged as that of the synchronous service request. The program counter (PC) associated with the wait is within EXE_STD$SYNCH_LOOP. When the kernel thread is placed back into execution, EXE_STD$SYNCH_LOOP repeats its status field test.

Figure 3.5 shows an example flow through $SYNCH and $SYNCH_INT. The program requests $QIO, specifying event flag 10 and an IOSB named MY_IOSB. The $QIO service clears event flag 10 and the status block, initiates the request, and returns to the program. Later the program requests $SYNCH, specifying event flag 10 and MY_IOSB.

Initially, the status is zero, and there is no user mode thread manager to notify, so EXE_STD$SYNCH_LOOP requests $SYNCH_INT. Holding the SCHED spinlock, EXE$SYNCH_INT tests the status. Finding it zero, it clears event flag 10 and places the kernel thread into an LEF wait.

At a later time, the previously requested I/O request completes. The special kernel mode AST associated with the completion of that request is queued to the waiting kernel thread, causing it to become computable. When the thread is placed back into execution, it executes the special kernel mode AST code, which sets local event flag 10 and stores status in the IOSB.

When the AST returns, the program resumes its execution of EXE_STD$SYNCH_LOOP. EXE_STD$SYNCH_LOOP repeats its test to determine whether the status is zero. This time, the status is nonzero, and EXE_STD$SYNCH_LOOP returns to its requestor.

### 3.6.5 Event Flag Wait States

There are two local event flag wait states, LEF and LEFO (local event flag outswapped), and two corresponding systemwide wait queue listheads (SCH$GQ_LEFWQ and SCH$GQ_LEFOWQ).

Only one common event flag wait state (CEF) exists for both resident and outswapped kernel threads. However, there is a separate common event flag wait queue listhead (see Figure 3.1) in each common event flag cluster. Each has the same overall structure as any other wait queue listhead (see Figure 3.2). Both resident and outswapped kernel threads waiting for flags in a common event flag cluster are queued to the same CEB wait queue. Having one queue in each CEB makes it easier to locate kernel threads whose wait is satisfied by the setting of a flag in that cluster.

The kernel thread waits at the access mode at which it requested the mode of caller wait service. The exact value of the PC saved for the waiting kernel thread varies with the particular wait or $SYNCH system service requested. In any case, the PC is defined to cause reexecution of the outer mode service whenever the kernel thread is placed into execution. If the kernel thread became computable because the wait condition is satisfied, it exits from the system service as long as the wait condition is still satisfied. If the kernel thread became computable as the result of AST enqueuing, at the completion of the AST it reexecutes the service and may be placed back into a wait. Section 3.9 discusses the implications of this, and Chapter 2 gives additional information.

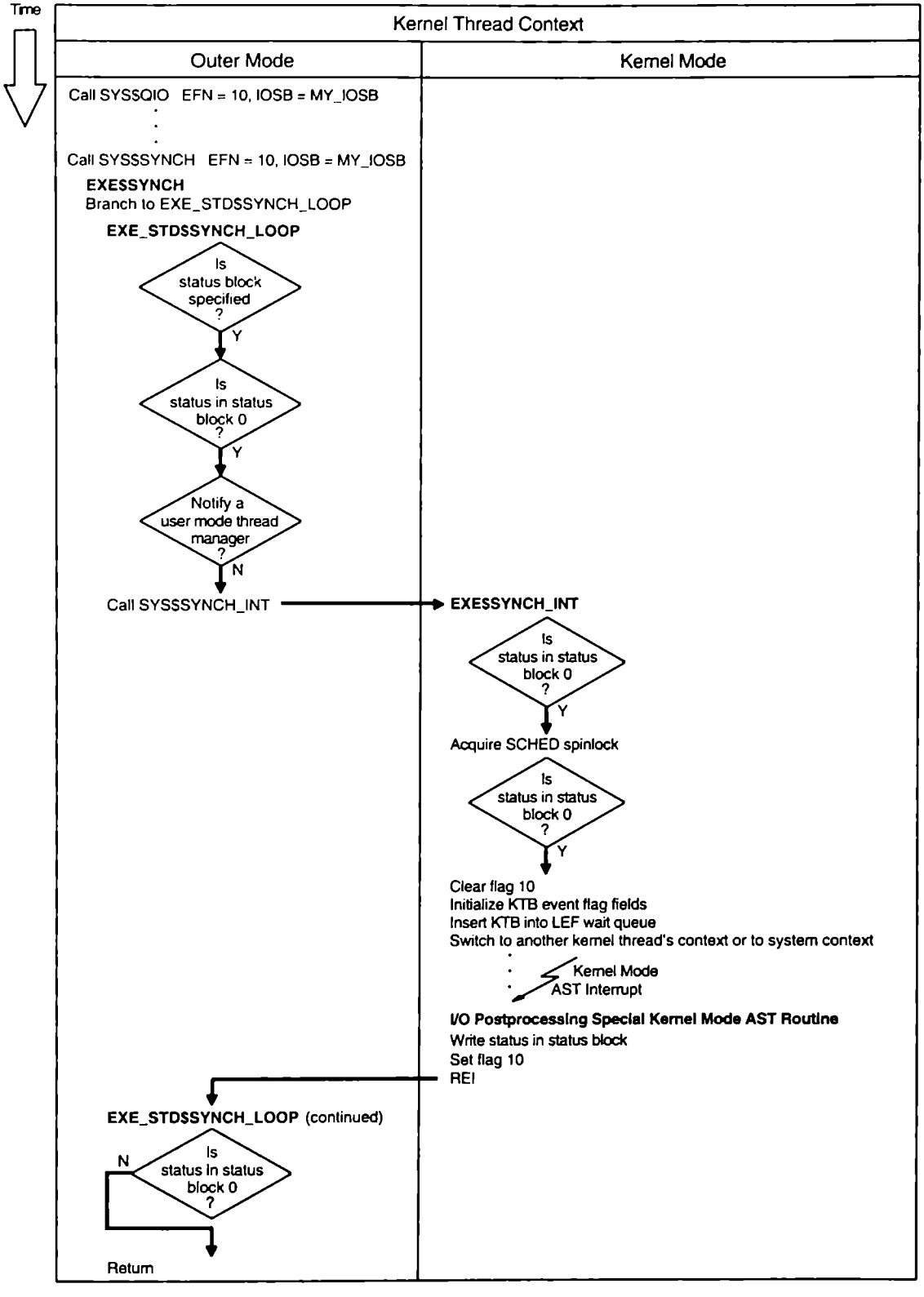



**Figure 3.5**
Example $SYNCH Flow

## 3.7 SETTING AN EVENT FLAG

A kernel thread sets an event flag directly by requesting the Set Event Flag ($SETEF) system service. A kernel thread can use this service at AST level to communicate with its mainline code. It can also use this service to set common event flags to communicate with other processes.

The executive sets event flags in response to I/O completion, timer expiration, the granting of a lock request, and completion of many other system services, such as Get Job Process Information ($GETJPI), Get Lock Information ($GETLKI), and Update Section ($UPDSEC).

The $SETEF system service and most other executive routines that set an event flag call the routine SCH$POSTEF, in module POSTEF. Its arguments are the number of the flag to be set, the IPID of the kernel thread in whose context that flag number is defined, and a priority increment class number (see Chapter 2). SCH$POSTEF performs the actual event flag setting and checks whether the event flag waits of any kernel threads in the same process are satisfied as the result of setting the flag.

SCH$POSTEF runs in kernel mode. It takes the following steps:

1. It first acquires the SCHED spinlock, raising IPL to IPL$_SCHED, to synchronize access to the scheduler database. (Note that the spinlock acquisition causes execution of an MB instruction, which orders any prior writes with the setting of the event flag.)
2. It then confirms that the process corresponding to the specified kernel thread still exists. If the process no longer exists, SCH$POSTEF releases the spinlock and returns the error status SS$_NONEXPR.
3. It checks that the event flag number is legal, returning the error status SS$_ILLEFC if the number is out of range.
4. It then determines what kind of event flag is being set: local, EFN$C_ENF, or common.
5. If a local event flag is being set, SCH$POSTEF takes the following steps:
   a. It sets the flag in the PHD cluster, if the process is resident, or in the PCB copy. It uses the load-locked/store-conditional mechanism to synchronize against concurrent writes to the flags. (For example, a kernel thread might be clearing one flag while I/O postprocessing code calls SCH$POSTEF to set another.)
   b. It tests whether the process is multithreaded and, if so, whether it is appropriate to notify a user mode thread manager. If it is, the service ORs the set flag into the process's flags accumulator at PCB$Q_POSTEF and queues an AST to notify the thread manager of the event-flag-setting event (see Chapter 5).
   c. It then checks the scheduling state of every kernel thread of the process to see whether the thread is in a LEF or LEFO state. If either, it checks whether this flag satisfies a wait request for that kernel thread. In the case of a $WFLOR wait, this flag merely has to match

one of the flags being waited for. For a $WFLAND wait, all the flags in the mask must be set to satisfy the kernel thread's wait request.

If a kernel thread's wait is satisfied, SCH$POSTEF reports an event-flag-setting event for the kernel thread by calling routine SCH$REPORT_EVENT, in module RSE.

d. SCH$POSTEF releases the SCHED spinlock and returns the success status SS$_WASSET or SS$_WASCLR, depending on the initial state of the flag.

This completes its processing for a local event flag.

6. If flag EFN$C_ENF is being "set," SCH$POSTEF takes the following steps:

   a. It tests whether the process is multithreaded and, if so, whether it is appropriate to notify a user mode thread manager. If it is, the service queues an AST to notify the thread manager of the event-flag-setting event.
   b. It checks the scheduling state of every kernel thread of the process to see whether the thread is in a LEF or LEFO state. If either, it checks whether the kernel thread is waiting for EFN$C_ENF. If so, a wait request for that kernel thread is satisfied, and SCH$POSTEF reports an event-flag-setting event for the kernel thread.
   c. SCH$POSTEF releases the SCHED spinlock and returns the success status SS$_WASSET.

   This completes its processing for an EFN$C_ENF event.

7. If a common event flag is being set, SCH$POSTEF takes the following steps:

   a. It first examines PCB$L_EFC2P or PCB$L_EFC3P, depending on the flag number, to see if the process has associated to a common event flag cluster. If not, it returns the error status SS$_UNASEFC.
   b. It gets the CEB address and sets the flag in the CEB using the load-locked/store-conditional mechanism to synchronize against concurrent writes to the flags.
   c. It scans the list of KTBs in the CEB wait queue to determine which, if any, of the kernel threads waiting for flags in this cluster has its wait request satisfied. SCH$POSTEF reports an event-flag-setting event for each such kernel thread.
   d. It releases the SCHED spinlock, restoring the previous IPL, and returns.

   This completes its processing for a common event flag.

When an event-flag-setting event is reported for a kernel thread in an event flag wait state, SCH$REPORT_EVENT changes its state to computable

resident (COM) or computable outswap (COMO) and, if appropriate, applies a priority boost, using the priority increment class number passed from SCH$POSTEF. SCH$REPORT_EVENT may request a rescheduling interrupt on behalf of the kernel thread or awaken the swapper process. Chapter 2 gives more details.

## 3.8 READING AND CLEARING EVENT FLAGS

The Read Event Flag (SREADEF) system service is informational. It has no effect on the computability of any kernel thread. The $READEF system service procedure, EXE$READEF in module SYSEVTSRV, runs in kernel mode. It uses its EFN argument to determine which cluster to read.

- For an EFN argument that identifies clusters 0 through 3, it copies the flags from either the PHD or the CEB that contains them to the location specified by its caller.
- For the EFN$C_ENF flag, which is always taken to be set, it stores a 32-bit mask of all zeros, except for the low-order bit, in the location specified by its caller.

It executes an MB instruction to order sensing the state of the flag with any subsequent actions the kernel thread might make based on its state. It returns the success status SS$_WASSET if any flag was set or if the EFN$C_ENF flag was specified as the EFN argument. Otherwise, it returns SS$_WASCLR, which is equal to SS$_NORMAL.

The Clear Event Flag (SCLREF) system service simply clears the event flag specified by its EFN argument. The $CLREF system service procedure, EXE$CLREF in module SYSEVTSRV, runs in kernel mode. It locates the cluster that contains the specified flag. Unless the specified flag is EFN$C_ENF, it clears the flag using the load-locked/store-conditional mechanism. It executes an MB instruction to order sensing the state of the flag with any subsequent actions the kernel thread might make based on its state. If the flag was EFN$C_ENF, it returns the success status SS$_WASSET. For any other flag, it returns the success status SS$_WASCLR or SS$_WASSET, depending on the initial state of the flag. It has no immediate effect on the scheduling state of any kernel thread.

## 3.9 CORRECT USE OF EVENT FLAGS

Using an event flag obtained through the Run-Time Library procedure LIB$GET_EF and released through LIB$FREE_EF (consult the *OpenVMS RTL Library (LIB$) Manual*) can help prevent inadvertent concurrent use of the same flag. Use of EFN$C_ENF with $SYNCH or synchronous system services is an alternative way to prevent concurrent use of the same flag.

It is strongly recommended that a unique status block be specified for each concurrent asynchronous operation and that $SYNCH, rather than

$WAITFR, be requested. As described in Section 3.6.3, $SYNCH determines completion based on the status block contents rather than on the state of the event flag. For improved performance, EFN$C_ENF should be specified as the flag. The status block contents should not be modified while the status block is in use.

For synchronous operations, a status block should be specified so that the implicit $SYNCH can check its status to determine completion. The status block contents should not be modified while the status block is in use.

With wait services rather than $SYNCH, a unique flag must be used for each operation. An event flag for which another thread is waiting should not be cleared.

The use of event flags 24 through 31 is reserved to Digital.

Section 3.6.5 describes the technique of waiting a kernel thread in the mode from which it requested $SYNCH or a wait service. This technique permits ASTs to be delivered to a kernel thread in an event flag wait, but it constrains the ways in which event flags can be used: flags for which a kernel thread is waiting should not be cleared by other threads of execution.

The result of clearing an event flag might be that a kernel thread becomes computable as the result of the flag's having been set but reenters the event flag wait state indefinitely when it reexecutes the event flag wait service and finds the flag no longer set. This could happen, for example, if kernel thread A waited for a common event flag set and then cleared by kernel thread B.

An indefinite wait is also possible when a local flag is used concurrently to signal multiple events. Consider the effect on the flow in Figure 3.4 of the following sequence of events:

1. As shown in the figure, the mainline code requests an asynchronous service and then waits for the associated local event flag to be set.
2. An outer mode AST is queued to the kernel thread, making it computable. (This AST is not in the figure.)
3. The AST procedure begins to execute.
4. The event associated with the mainline code's service occurs, and the local flag is set.
5. The AST procedure requests an asynchronous service, specifying the same flag.
6. The service procedure, in preparation for a possible wait request, clears the flag; initiates the requested task; and returns to the AST procedure, which also returns.
7. The mainline code repeats its event flag wait test. Because the flag has been cleared, the mainline code reenters its wait state.

The previously described constraint (that a flag for which a thread is waiting should not be cleared by other threads of execution) applies to all wait services but has additional significance for the $WFLAND system service.

The $WFLAND system service generates a wait mask based on the input mask flags that are not already set at the time the service is requested. However, each time the kernel thread is placed back into execution as a result of AST delivery, the kernel thread reexecutes the $WFLAND service and, each time, the event flag wait mask is built anew. No record is kept that some of the flags have been set and should not be waited for again if the service is reexecuted.

## 3.10 EXAMPLE OF INTERPROCESS SYNCHRONIZATION THROUGH COMMON EVENT FLAGS

The use of common event flags is one method of interprocess synchronization. A kernel thread in one process can reach a critical point in its execution and wait for a common event flag. A kernel thread in another process can enable the first kernel thread to continue its execution by setting the flag.

A common event flag can also be used as a semaphore to gain access to a resource shared among processes. One such application is outlined here. It first requires creation of a common event flag cluster with all its flags set. Each flag can be used as an individual lock. Each cooperating process must associate to the common event flag cluster.

Before any kernel thread in a process uses the resource represented by a particular event flag, it must execute a sequence such as the following, which uses event flag number 65 as an example:

```
5$:   $CLREF_S    EFN=#65          ;Clear the event flag
      CMPL        R0,#SS$_WASSET   ;Was its previous state = 1?
      BEQL        10$              ;Branch if yes
      $WAITFR_S   EFN=#65          ;Else wait for flag
      BRB          5$
10$:                               ;Proceed to access resource
      .
      .
      .
      $SETEF_S    EFN=#65          ;Set the event flag
```

Clearing an event flag is an interlocked operation implemented by the executive. Only one kernel thread at a time can clear the flag and cause the transition in its state from set to clear. That kernel thread then "owns" the flag and its associated resource. Any other kernel thread that clears the flag receives a was-clear status and must wait for the flag to be set.

The kernel thread that owns the flag can then access the resource without synchronization problems. When the kernel thread's accesses to the resource are complete, the kernel thread sets the flag, relinquishing ownership of the flag and resource. The kernel threads that were waiting for the flag are made computable and repeat their attempts to cause the event flag transition from set to clear.

## 3.11 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIB]CEBDEF.SDL
[LIB]PCBDEF.SDL
[SYS]POSTEF.MAR
[SYS]SYSASCEFC.MAR
[SYS]SYSEVTSRV.MAR
[SYS]SYSSYNCH.MAR
[SYS]SYSWAIT.MAR

# 4 Lock Management

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
Shakespeare, *Hamlet*, I, iii

This chapter provides a brief overview of locks and lock-related terminology. It describes lock management data structures, lock management system services, resource domains, and deadlock detection.

The treatment in this chapter assumes that the reader is familiar with the description of the OpenVMS lock management system services found in the *OpenVMS System Services Reference Manual*. Although VMScluster distributed lock management is briefly discussed here, its details are beyond the scope of this book.

## 4.1 OVERVIEW

Lock management system services provide a set of primitives that enable cooperating processes to synchronize their accesses to global sections, files, or any other entities. Through these services, a process assigns a name to an entity, requests a lock on the name, and restricts its accesses to the entity based on a locking convention. The services do not maintain a physical link between the named resource and an actual entity. Processes requiring synchronized access to an entity must agree upon the resource name that represents the entity and must explicitly cooperate by locking this resource name.

To prevent interference between applications, a resource name is only meaningful within a resource domain. The same resource name in two different domains represents different resources. Access to a resource domain is controlled by normal security policy: owner, protection mask, and access control list (ACL). A resource domain is identified by number. By default a process may take locks on resources within the domain that corresponds to its user identification code (UIC) group number. Section 4.2.3 provides more information on resource domains.

A lock is characterized by its mode, that is, the extent of the shared access to the resource that it allows.

Locks that permit mutual shared access are termed compatible. Processes holding compatible locks on a resource have concurrent access to the entity it represents. A kernel thread requesting an incompatible lock on a resource is denied access. Optionally, such a kernel thread can be placed into a wait state until blocking locks are released and the resource becomes available.

A lock request is issued from the context of a kernel thread. Any asynchronous system traps (ASTs) associated with the lock request are delivered to that kernel thread, and if the lock request must wait, it is the kernel thread, not the process, that is placed into a wait state. However, the resulting lock is owned by the process rather than by any individual kernel thread. Any of the process's kernel threads can convert, cancel, or dequeue the lock. Chapter 5 describes kernel threads.

A lock request exists in one of three states: granted, granted but awaiting conversion to another mode, or waiting to be granted. A kernel thread can acquire a new lock, convert a granted lock to another mode, cancel a waiting lock request, cancel a lock conversion, or dequeue a granted lock.

Lock management system services determine which lock requests to grant based on the compatibility of the request with currently granted locks and on the presence of waiting or converting lock requests. Section 4.2.8 summarizes this behavior, and Section 4.3.3 provides details.

The following system services, described in Section 4.3, allow a kernel thread to enqueue and dequeue lock requests, obtain information about locks, and join resource domains:

- Enqueue Lock Request [and Wait] (SENQ[W])
- Dequeue Lock Request (SDEQ)
- Get Lock Information [and Wait] (SGETLKI[W])
- Set Resource Domain (SSET_RESOURCE_DOMAIN)

In response to the first request to lock a given resource, the executive creates a data structure called a resource block, described in Section 4.2.1, to represent the resource. It grants the requestor access to the resource. When a subsequent request is made to lock the same resource, the executive uses the information maintained in the resource block to determine whether the requested lock can be granted.

A lock block data structure, described in Section 4.2.4, represents a lock request. Each lock block for a given resource can be found in one of three queues located in the resource block. When a lock request is granted, its lock block is placed on the resource's granted queue. A lock granted in one mode can later be converted to another mode; a lock block representing a conversion request incompatible with currently granted locks is placed on a resource's conversion queue. A lock block representing a new request for incompatible access to a resource, or a new request that arrives while the conversion or wait queues are populated, is placed on the resource's wait queue. Section 4.3.3 describes lock-granting behavior in detail.

When two processes compete for incompatible access to a resource, only one gets access; the other process's request is not granted until the competing process releases the lock or converts the lock to a mode compatible with the other's request. Perhaps inadvertently, application-locking protocols can cause two or more competing processes to wait for each other to

release a lock. Such waits can never be satisfied. This condition is called a deadlock. The executive does not prevent applications from creating deadlock conditions. It does, however, detect and break deadlocks, as described in Section 4.4.

When a process requests the creation or conversion of a lock, it can include the address of a blocking AST routine. When such a request is granted, the lock management code checks whether the lock is blocking the granting of any locks on the conversion or wait queues. If so, it calls the blocking AST routine. The actions of the blocking AST routine are determined by the application-locking protocol; typically, it would convert the lock to a less restrictive mode. Some application-locking protocols use blocking ASTs to avoid deadlocks.

A resource can include a value block, a 16-byte field stored in the resource block. A lock request from a kernel thread can obtain a private copy of the value block when the lock is granted and when it is converted to a mode at least as restrictive as its current grant mode. To write new information to the value block, a process holding a PW or EX mode lock (see Table 4.1) can dequeue the lock or convert the lock to an equally restrictive or less restrictive mode.

Lock management is central to VMScluster operation. Lock management system services provide clusterwide synchronization to OpenVMS facilities and user applications. Appendix A describes the manner in which some of the facilities define resources and use locks.

A resource's identity is clusterwide in scope. Resource blocks and lock blocks, as well as other lock management data structures, are distributed among the nodes of a VMScluster system. The node responsible for granting locks on a given resource is called its resource master. All lock requests for the resource must be communicated to this master node, since it maintains the list of granted and waiting lock requests for the resource. Under the conditions outlined in Section 4.2.7, mastership can move from one node to another; that is, the resource can be remastered.

A system application called the distributed lock manager assists the lock management system services in maintaining the clusterwide lock database. Participating cluster nodes share lock management overhead while keeping internode traffic at a minimum level. A detailed description of the distributed lock manager is outside the scope of this book. Its basic operation is presented in Section 4.2.7.

## 4.2 LOCK MANAGEMENT DATA STRUCTURES

The lock database consists of the following kinds of structures:

- Resource blocks (RSBs), which represent the entities for which locks have been requested
- One resource hash table, which locates the RSBs

- Lock blocks (LKBs), which describe locks requested by processes
- One lock ID table, which locates the LKBs

### 4.2.1 Resource Blocks

An RSB is allocated whenever a process requests the $ENQ system service specifying a resource not already defined. It remains allocated as long as there is at least one lock on the resource.

A resource is uniquely identified by the following combination:

- Resource name string of 1 to 31 characters
- Access mode
- Resource domain: an explicitly specified domain, the process default domain, or the system domain. Section 4.2.3 describes resource domains.
- Address of parent RSB, if any

Two resources with identical resource name strings are different unless their resource domains, access modes, and parents match. (Since a subresource inherits its resource domain and access mode from its parent, the name string and parent are its two determining factors.)

Resources can be hierarchical. For example, a resource can be defined to represent a particular file, with subresources for particular records in the file. The file resource is a parent resource to the resources representing records in the file. A record subresource may be a parent resource to additional subresources that represent fields in the record. The combination of a resource and all its subresources is called a resource tree. The top-level resource in the tree, the one with no parent, is called the root resource.

The first lock request for a resource can specify the parent of the resource, thereby defining its relationship in a tree. If no parent is specified, the resource becomes a root resource. To synchronize access to a subresource, a process must first lock the root resource and then each subresource in the branches of the resource tree leading down to the target subresource, including the target subresource. For example, before locking a field in a record, a process must lock the associated file and then the record.

Note that the OpenVMS executive enforces this policy and returns an error if a process tries to acquire a lock on a subresource without holding a lock on the parent resource.

The root resource list, whose listhead is the global symbol LCK$GL_RRSFL, links the root resources known by the local system. Subresources are linked to these root RSBs. The maximum depth of a resource tree is 127.

Figure 4.1 shows the layout of an RSB. RSB$T_RESNAM and RSB$B_RSNLEN contain the resource name string and its length. RSB$W_GROUP contains the resource domain (a UIC group), RSB$B_RMOD the access mode, and RSB$L_PARENT the address of the parent RSB. Together, these five

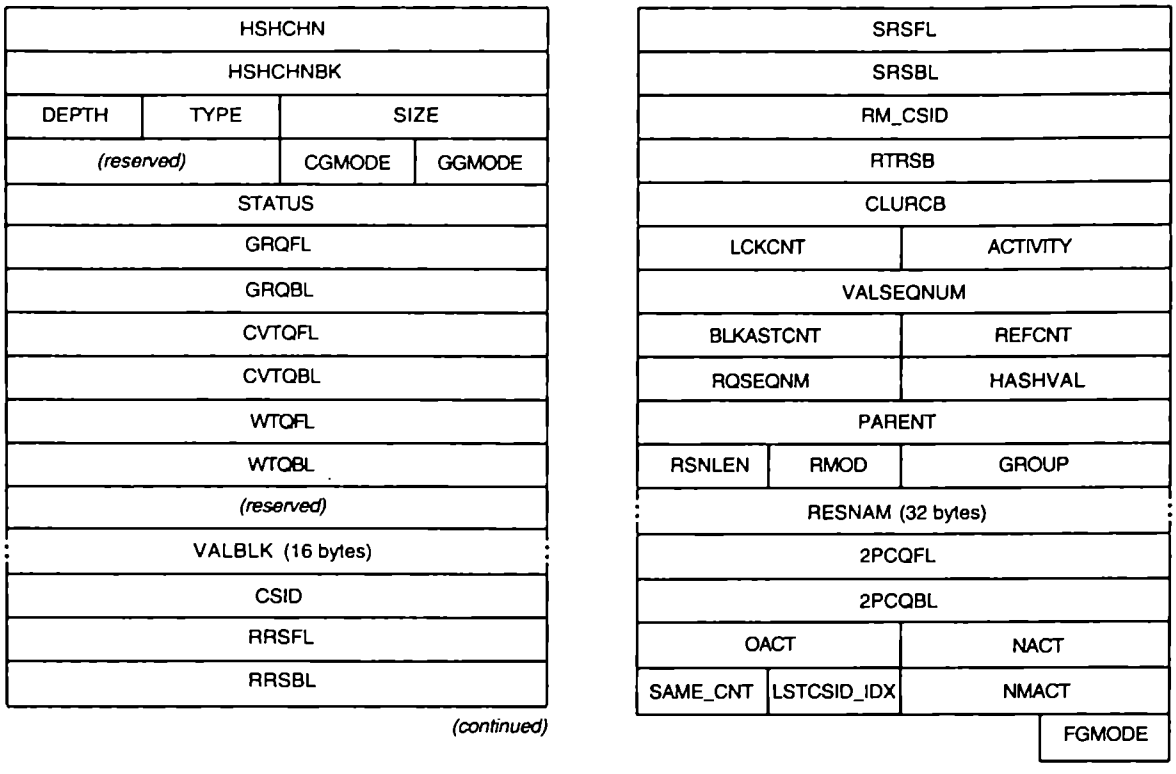



**Figure 4.1**
Layout of a Resource Block (RSB)

fields uniquely identify a particular resource. The first lock request for a root resource establishes the resource domain and the access mode of the resource tree. RSB$W_GROUP and RSB$B_RMOD propagate from the root resource to all subresources.

RSB$B_DEPTH indicates the position of the resource in a resource tree; a root resource has a depth of zero. The depth of a subresource is set to 1 more than its parent's RSB$B_DEPTH. Root resources are linked to form a queue through their RSB$L_RRSFL and RSB$L_RRSBL fields. All subresources of the root resource are linked to form a queue through the fields RSB$L_SRSFL and RSB$L_SRSBL. Each subresource contains the address of its root RSB in RSB$L_RTRSB; a root resource contains its own address. Figure 4.2 shows this linkage of root and subresources. RSB$W_ACTIVITY tracks the local node's use of the resource; a root resource with a low value is more likely to be remastered (see Section 4.2.7).

If the resource has a parent resource, its access mode is taken from the parent. Otherwise, access mode is set to the access mode from which the $ENQ request was issued, or to the $ENQ system service argument ACMODE, whichever is less privileged. The resource's access mode qualifies the name space in which the resource exists. It specifies the least privileged mode from which locks can be queued to the resource and from which information

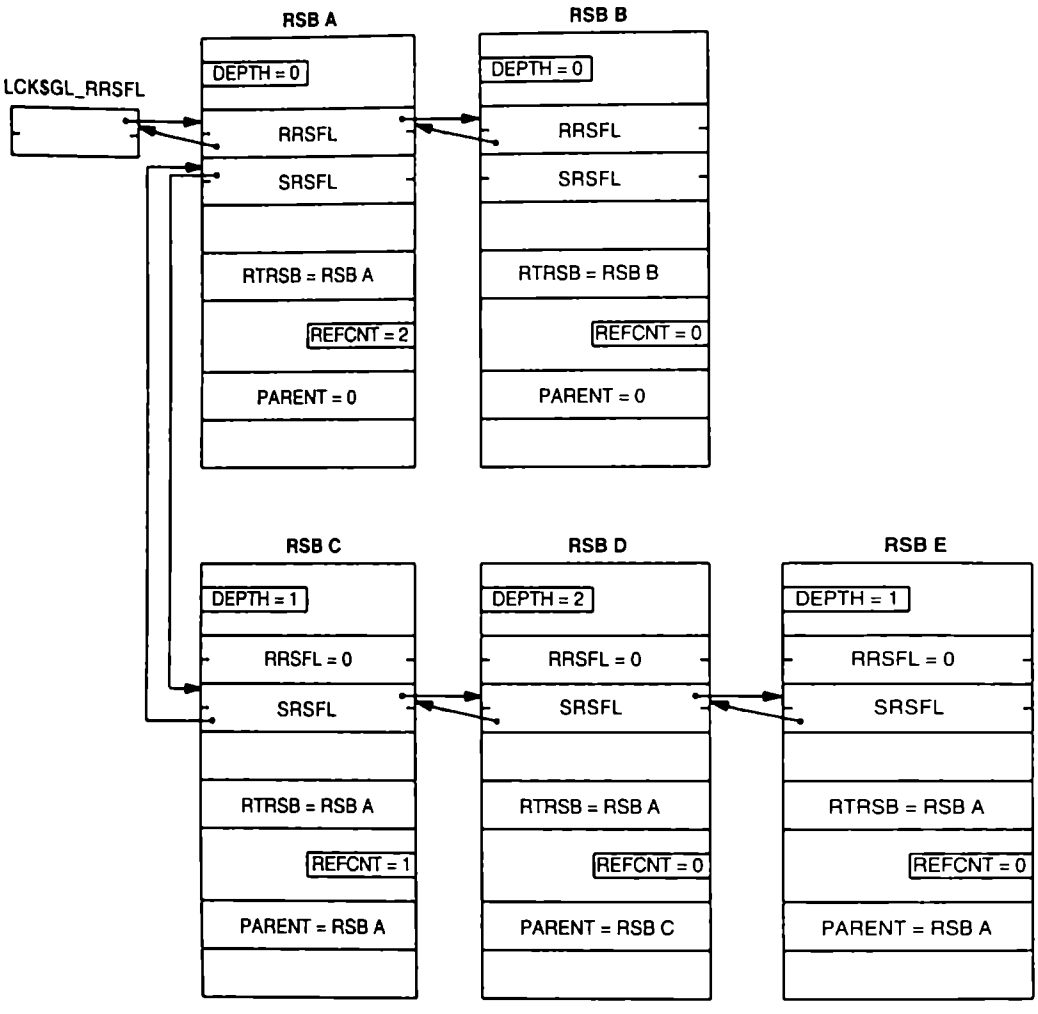



**Figure 4.2**
Root Resources and Subresources

about the locks can be obtained. In a parent RSB, RSB$W_REFCNT counts the number of its immediate subresources.

An RSB contains listheads for the granted, conversion, and wait queues of LKBs associated with the resource. The listhead for the granted LKB queue consists of the fields RSB$L_GRQFL and RSB$L_GRQBL. The listhead for the conversion queue is the fields RSB$L_CVTQFL and RSB$L_CVTQBL. The listhead for the wait queue is the fields RSB$L_WTQFL and RSB$L_WTQBL. Section 4.2.4 describes the significance of these queues.

An RSB also contains 16 bytes that form the value block for the resource at the field RSB$Q_VALBLK. RSB$L_VALSEQNUM contains the sequence number associated with the contents of the value block.

Other RSB fields are described in later sections of this chapter.

RSBs are allocated from nonpaged pool. For enhanced performance, preallocated RSBs are cached. The routine that allocates RSBs first attempts to

remove one from an RSB cache list at global location LCK$GL_RSB_HEAD. If the cache is empty, the routine allocates an RSB from nonpaged pool. When an RSB is no longer needed, it is put into the cache. To ensure that the cache does not grow out of proportion, a routine periodically trims the cache by deallocating excess RSBs back to nonpaged pool.

### 4.2.2 Resource Hash Table

The resource hash table locates all the RSBs in use. The combination of the resource name string and its length, resource access mode, domain, and parent RSB hash value is hashed and the result stored in RSB$W_HASHVAL. The hashing algorithm is similar to the algorithm used for hashing logical names, described in Chapter *Logical Names*. The contents of RSB$W_HASHVAL index a particular entry in the resource hash table. More than one resource name can hash to the same value. Each longword entry in the hash table is either zero or a pointer to a list of RSBs with that hash value. If a longword entry in the resource hash table contains a zero, there is no RSB with that hash value.

Because the RSBs are maintained in a list that is doubly linked but not circular (the resource hash table itself contains no backward pointers), the list of RSBs is termed a chain. The first two longwords in each RSB contain the forward and backward pointers for the resource hash chain. The last block in each chain has a zero forward pointer.

Figure 4.3 shows the structure of the resource hash table and its relation to hash chains.

The resource hash table is allocated from nonpaged pool. The global location LCK$GL_HASHTBL contains its address. The number of longword entries in the resource hash table is determined by the SYSGEN parameter RESHASHTBL. Note that the parameter does not limit the number of RSBs that can be created. However, the combination of a small hash table and many RSBs can result in longer hash chains than might be desirable for optimal performance, since the hash chain is searched sequentially.

### 4.2.3 Resource Domains

In prior versions of OpenVMS, a resource name was qualified by the UIC group of the creating process (or by the UIC group of the parent resource if one existed). To share a resource, processes had to be members of the same UIC group. That is, the name space, or scope of the resource name, was the UIC group.

The single exception was the system name space; specifying the LCK$V_SYSTEM flag when requesting the $ENQ service caused the UIC group to be ignored in the resource name. In effect, such resources shared a UIC group of 0. However, only a process executing in kernel or executive mode, or holding the SYSLCK privilege, could create such a lock.

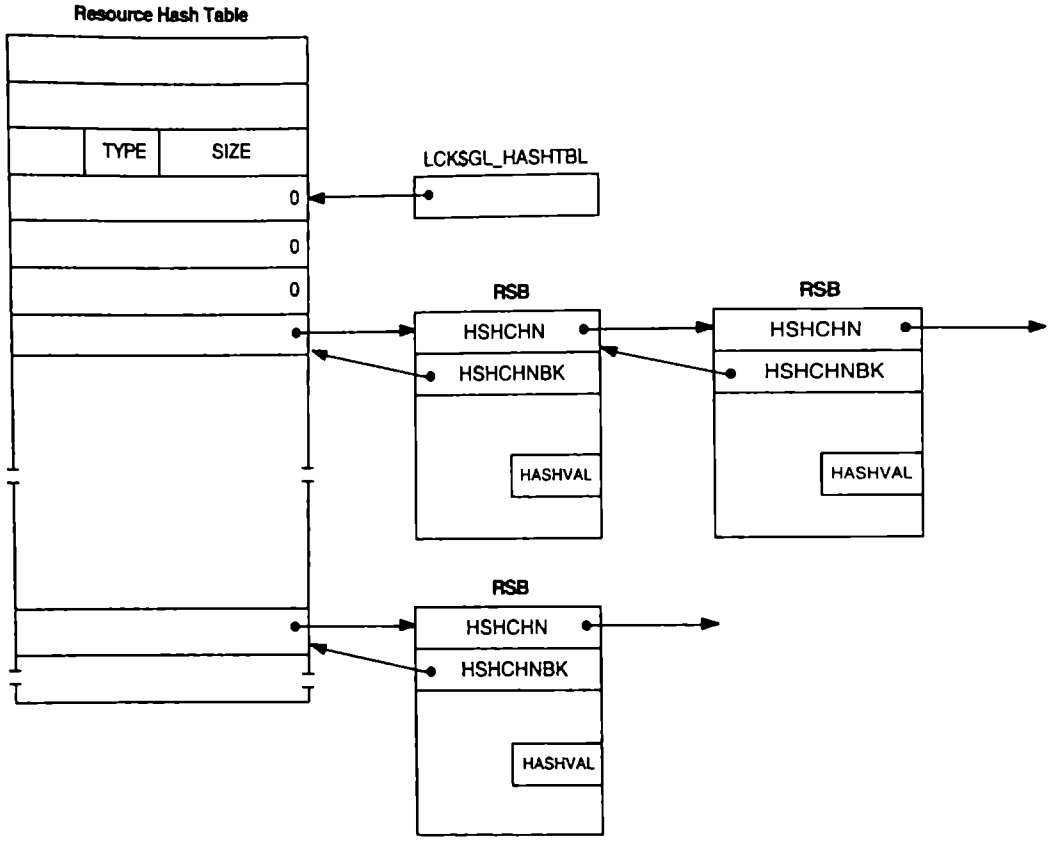



**Figure 4.3**
Resource Hash Table and Hash Chains

Thus, a process could previously access only two resource domains—its current UIC group and and the system domain. The introduction of extended resource domains allows processes from different UIC groups to share a name space by explicitly joining a resource domain and specifying that resource domain in a $ENQ request.

A process's association with a resource domain determines its right to create locks on a resource name, to read the value block, and to write the value block. A process joins a resource domain by requesting the $SET_RESOURCE_DOMAIN system service, described in Section 4.3.8. $SET_RESOURCE_DOMAIN validates the request within the security policy and returns a resource domain ID, RSDM_ID, which the process uses as input to the $ENQ system service. To join a resource domain, the process must hold an identifier that grants access to that resource domain.

The current implementation of resource domains is primarily designed to provide access protection for the resource value block. Resource domains are still UIC-based, the system name space is still available and still requires the SYSLCK privilege, and the default name space is still the UIC group. How-

ever, the option of extending name spaces by explicitly granting access to a name space to processes from different UIC groups provides some additional design flexibility as well as value block protection.

Each process is associated with the system domain at login. A process must be associated with the system domain to access the value blocks of system locks, used by the file system on the process's behalf. The SYSLCK privilege still determines a process's ability to create a system lock; the default association just allows access to the value blocks of system locks.

A process is also associated with a default process resource domain. Its association is established the first time it requests the $ENQ system service to create a lock. The default process resource domain is determined by the process's UIC group at the time of this $ENQ request. If the process later changes its UIC group, for instance, through the DCL command SET UIC, its next $ENQ request to create a lock causes the default process resource domain to change as well, reflecting the new UIC.

The $ENQ system service stores the resource domain ID and value block access bits in the LKB, and the UIC group and access mode in the RSB. For a root lock, it obtains the resource domain ID and value block access bits from the default process resource domain, unless the $ENQ request specifies otherwise. It uses the resource domain ID to locate the resource domain data block (RDDB) and obtains the UIC group and access mode from the RDDB. Figure 4.4 shows the layout of the RDDB and two related structures, the resource domain pointer block (RDPB) and the resource domain access block (RDAB).

An RDDB describes an active resource domain; an RDDB for each active domain is linked on the system listhead EXE$GQ_RSDM_LIST. RDDB$L_GROUP contains the UIC group number associated with the domain. RDDB$L_ORB points to an object rights block (ORB) that contains the resource name, protection information, and pointers to any ACLs. When a new RDDB is created, the AUDIT_SERVER process ensures that the RDDB correctly represents the object security profile for the resource domain. The DCL command SET SECURITY establishes this profile, which is maintained across reboot and shared clusterwide.

An RDAB associates the process with the resource domain. RDAB$L_RDDB locates the domain's RDDB, RDAB$L_SEQ counts the number of reuses of the RDAB, and RDAB$L_ACCESS identifies the access allowed to the domain (read, write, or lock). RDAB$L_ACMODE is the least privileged mode allowed to access the domain, and RDAB$L_DOM_ACMODE is the most privileged mode allowed access.

A process's RDPB list contains its default domain information (for faster access) and an RDAB for each resource domain that the process has joined. In the RSDM_ID returned by $SET_RESOURCE_DOMAIN, the low word is an

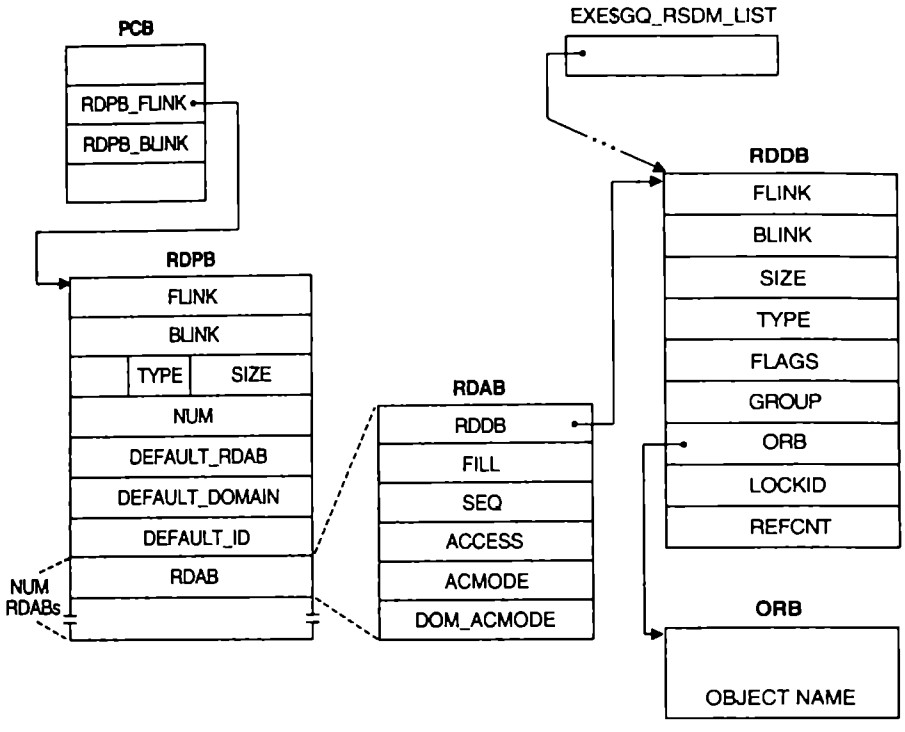



**Figure 4.4**
Resource Domain Data Structures

index into the RDPB that locates an RDAB; that RDAB's SEQ value should match the high word of the RSDM_ID.

### 4.2.4 Lock Blocks

The $ENQ system service allocates an LKB in response to a process's first lock request on a resource. The LKB is assigned a unique lock ID used to identify the lock in subsequent lock conversion or dequeue requests. The LKB contains both the process ID of the creator process and the thread ID of the kernel thread that requested the $ENQ system service. A lock is owned by a process, not by an individual kernel thread. Any of the process's kernel threads can convert or dequeue the lock; any ASTs associated with the lock request are delivered to the kernel thread whose thread ID is contained in the LKB, that is, the kernel thread that most recently converted the lock. Chapter 5 describes thread IDs and AST delivery to kernel threads. When a process dequeues a lock, the LKB is deallocated. When a process is deleted, all its locks are dequeued. Figure 4.5 shows the layout of a lock block.

Table 4.1 lists the lock modes, their meanings, and the other granted lock modes with which each lock is compatible. A lock granted at one mode can later be converted to another mode. LKB$B_RQMODE specifies the requested lock mode of the lock, and LKB$B_GRMODE, the granted lock mode.

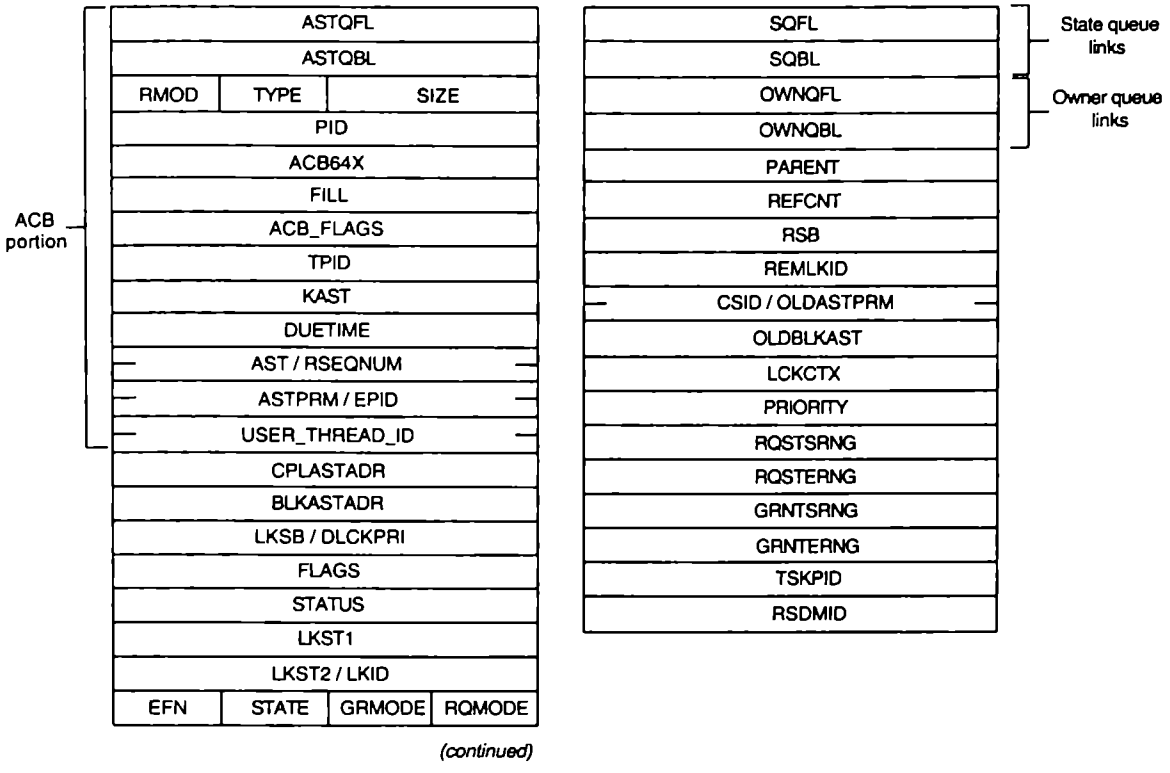



**Figure 4.5**
Layout of a Lock Block (LKB)

Table 4.1 Meaning and Compatibility of Lock Modes

| *Mode of Requested Lock* | *Meaning* | *Compatible with* |
|---|---|---|
| NL | Null—grants no access; used as an indicator of interest or a placeholder for future lock conversion | NL, CR, CW, PR, PW, EX |
| CR | Concurrent read—grants read access and allows resource sharing with other readers and writers | NL, CR, CW, PR, PW |
| CW | Concurrent write—grants write access and allows resource sharing with other writers | NL, CR, CW |
| PR | Protected read—grants read access and allows resource sharing with other readers, but not writers | NL, CR, PR |
| PW | Protected write—grants write access and allows resource sharing with CR-mode readers, but not writers | NL, CR |
| EX | Exclusive—grants write access and prevents resource sharing with any other readers or writers | NL |

A lock can be granted, converting, or waiting, depending on the lock modes of other locks on the resource. A new lock is granted and its LKB placed on the RSB granted queue if its lock mode is compatible with those of locks already granted on the resource and if the conversion and wait queues are empty. Otherwise, it is placed at the end of the wait queue. A subsequent attempt to convert a granted lock to a more restrictive lock mode can result in the insertion of its LKB at the end of the conversion queue. Conversion requests have precedence over all waiting requests and new lock requests. Waiting requests have precedence over new lock requests. Section 4.2.8 summarizes lock-granting behavior.

LKB$B_STATE specifies the current lock condition, for example, granted, waiting, or in a conversion queue. LKB$L_SQFL and LKB$L_SQBL link the LKB into the appropriate state queue in its RSB. LKB$L_RSB contains the resource address. Typically, a lock in the conversion or wait queue is also queued to the lock timeout queue through the fields LKB$L_ASTQFL and LKB$L_ASTQBL. If the lock request is not granted within a certain amount of time, a deadlock search is triggered (see Section 4.4.1).

A lock with a parent lock and resource is termed a sublock. An LKB describing a sublock contains the address of the parent LKB in field LKB$L_PARENT; the parent LKB has no corresponding pointer to the sublock. The RSB associated with the sublock points to the parent resource through the field RSB$L_PARENT; the parent resource has no corresponding pointer to the subresource. These relations are shown in Figure 4.6. LKB$W_REFCNT specifies how many sublocks have that LKB as their parent. Although a longword of storage is reserved for the LKB REFCNT field, the field is referenced as both LKB$W_REFCNT, by earlier code, and as LKB$L_REFCNT.

The first part of an LKB is an AST control block (ACB). When a lock request is granted, the LKB/ACB can be queued to the process's process control block (PCB) or the kernel thread's AST queues through the fields LKB$L_ASTQFL and LKB$L_ASTQBL. Queued as an ACB, it describes a special kernel mode AST, a blocking AST, or a completion AST (see Section 4.3.3).

LKB$L_PID contains the internal process ID of the process that requested the lock. LKB$L_TPID contains the internal thread ID of the kernel thread that requested the lock. Since validation of LKB ownership is based on the process ID, all of a process's kernel threads can access a lock requested by any kernel thread. Including the thread ID ensures that ASTs are delivered to the correct kernel thread. Chapter 5 describes thread IDs and AST delivery to kernel threads.

LKB$B_RMOD specifies the access mode at which completion and blocking ASTs for this lock are delivered. The access mode from which the $ENQ system service is requested, rather than an $ENQ service argument, determines the value of LKB$B_RMOD. This field also specifies the least privileged access mode from which the lock can be converted or dequeued. If a

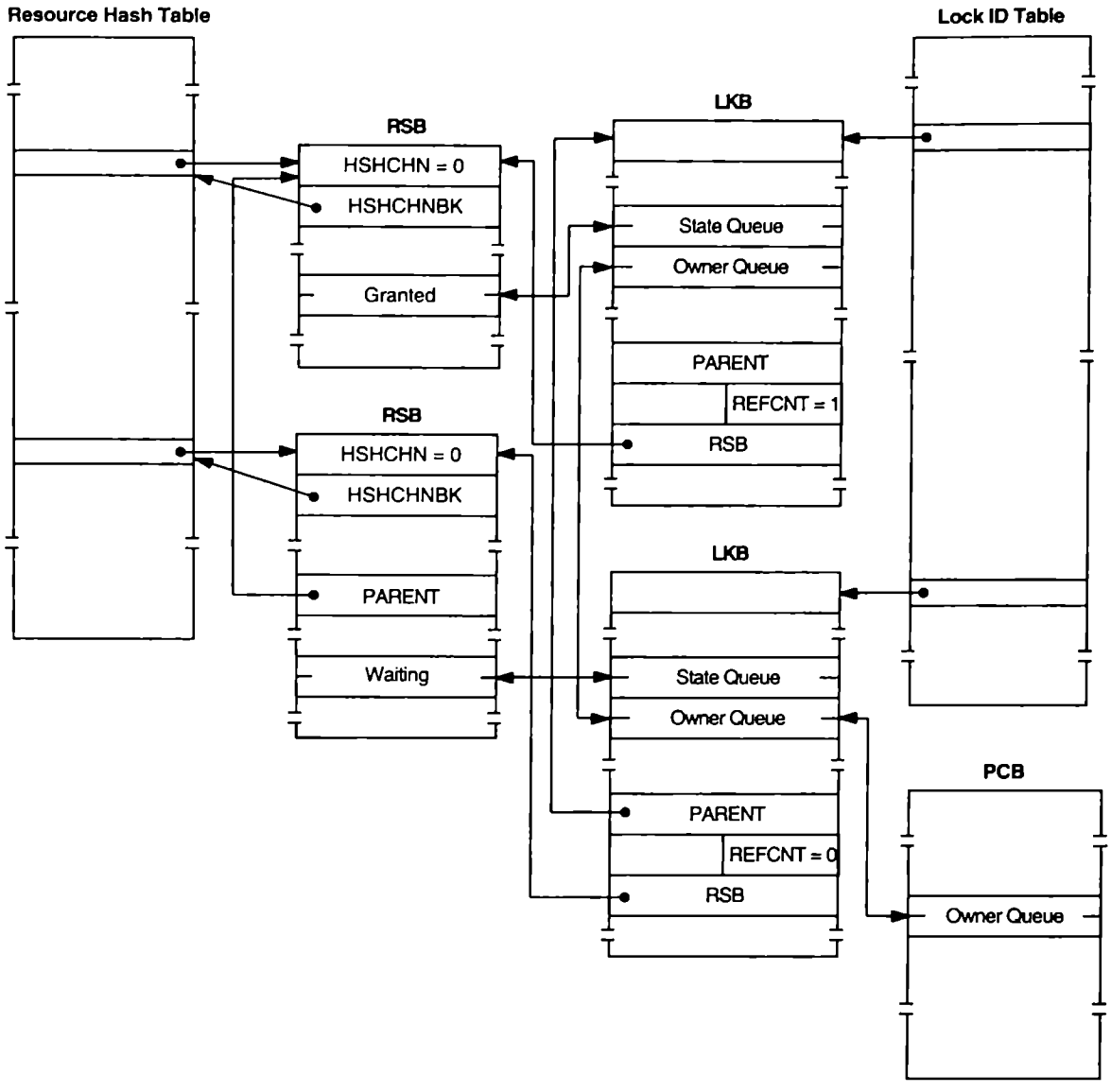



**Figure 4.6**
Relations Between Locks and Sublocks

lock has a parent, the lock's access mode must not be more privileged than that of its parent.

LKB$L_EPID contains the extended process ID (see Chapter 7) of the process that requested the lock. LKB$L_CPLASTADR and LKB$L_BLKASTADR contain the procedure values of the completion and blocking AST procedures requested by the process. LKB$L_LKSB contains the address of the request's lock status block (LKSB). LKB$L_LKST1 contains the condition value to be copied to the lock status block. The second longword of lock status, LKB$L_LKID, contains the lock ID.

Other LKB fields are described in later sections of this chapter.

LKBs are allocated from nonpaged pool. For enhanced performance, preallocated LKBs are cached. The routine that allocates LKBs first attempts to remove one from an LKB cache at global location LCK$GL_LKB_HEAD. If the cache is empty, the routine allocates an LKB from nonpaged pool. When an LKB is no longer needed, it is put into the cache. To ensure that the cache does not grow out of proportion, a routine periodically trims it by deallocating excess LKBs back to nonpaged pool.

### 4.2.5 Lock ID Table

The lock ID table, shown in Figure 4.7, contains a pointer to each LKB. A lock ID consists of an index into the lock ID table and a sequence number identifying this particular use of that index. When a lock index is in use, its entry in the lock ID table contains the address of the associated LKB.

The entry for an unused index has two pieces of information. The high-order byte contains the updated sequence number for that index. The low-

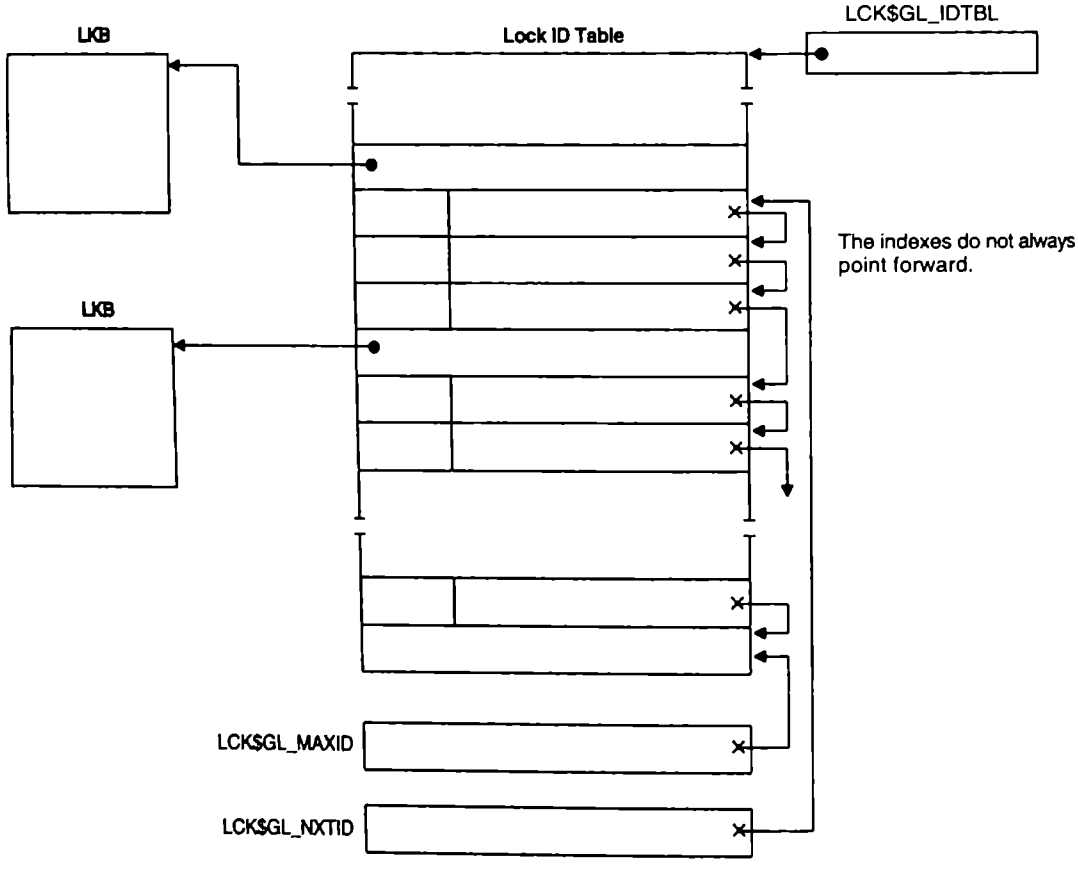



**Figure 4.7**
Structure of the Lock ID Table

order three bytes contain the index of the next unused entry in the lock ID table. The unused entries in the lock ID table are thus linked together, with the listhead at global location LCK$GL_NXTID. When a new lock is requested, its index is taken from LCK$GL_NXTID, which is updated to point to the next unused entry. Global location LCK$GL_LSTFREE contains the index of the last free lock ID.

Because it is possible for an erroneous value to be passed as a lock ID to a lock management system service, the system services validate the lock ID. They compare the requestor's process identification (PID) and access mode with the PID and access mode stored in the LKB. The PIDs must match and the requestor's access mode must be at least as privileged as that of the lock. If the comparison fails, the service exits with the error status SS$_IVLOCKID.

The global symbol LCK$GL_IDTBL points to the lock ID table. The SYSGEN parameters LOCKIDTBL and LOCKIDTBL_MAX control the size of the lock ID table. LOCKIDTBL_MAX specifies the maximum size of the lock ID table and thus the maximum number of locks. LOCKIDTBL specifies the initial size of the lock ID table.

During system initialization enough system page table entries (SPTEs) are allocated to map LOCKIDTBL_MAX + 256 longwords for lock IDs. Physical pages are allocated for LOCKIDTBL + 256 longwords and mapped through the first SPTEs. All remaining SPTEs map a zeroed physical page. Global location LCK$GL_LKIDFREE keeps a running count of free lock IDs, and LCK$GL_LKIDCNT, the total number of lock IDs. LCK$GL_NXTID is initialized to contain the index of the first entry in the table. Each lock ID is initialized to contain the index to its successor, starting with the first one. LCK$GL_MAXID contains the index to the last entry in the lock ID table.

When a new lock is requested and no lock ID is free (LCK$GL_LKIDFREE is zero), the lock ID table must be expanded unless it already contains LOCKIDTBL_MAX entries. To expand the table, the executive allocates a physical page and maps it with the first free SPTE. Each longword in that page represents a new lock ID. Starting with the first one, the executive initializes each new lock ID to contain the index to its successor. The executive adjusts the last entry in the lock ID table to contain the index of the first entry in the new page, and updates LCK$GL_MAXID, LCK$GL_LKIDFREE, and LCK$GL_LKIDCNT.

A lock to be dequeued is identified by its lock ID. The lock ID locates the corresponding lock ID table entry. The table entry has the address of the LKB to be deallocated. When the LKB is deallocated to the LKB cache, its lock ID is, in effect, cached as well. At deallocation, the corresponding lock ID sequence number byte is incremented, generating a new lock ID with the same lock ID table index. If the sequence number wraps (if it exceeds 255), the resulting lock ID is the same lock ID that was generated the first time this lock ID table index was used. This lock ID is not reused now; it is retired

to the end of the free lock ID list, at LCK$GL_LSTFREE. A new lock ID, obtained from LCK$GL_NXTID, is associated with the LKB and the lock ID table entries for both lock IDs are updated. The retired lock ID will not be reused until all others on the free list have been used.

By adding 256 to LOCKIDTBL and LOCKIDTBL_MAX and not including the additional count in LCK$GL_LKIDFREE and LCK$GL_LKIDCNT, the executive guarantees that there will always be at least 256 entries in the chain of free lock IDs, each capable of 256 uses before recycling. Thus, although the lock ID sequence number is only a byte field, a lock ID is reused only after at least 65,535 others (or 256 × 256) have been used.

When LCK$GL_LKIDFREE goes down to zero, there are still 256 free lock IDs in the chain, although they cannot be used until the lock ID table is expanded. If the lock ID table can no longer be expanded, the executive generates the fatal bugcheck LOCKMGRERR, even though, at this point, there would still be 256 free lock IDs. Since the lock ID index field is three bytes, the theoretical maximum is 16,777,216 locks, although the current maximum is 4,194,304 locks.

### 4.2.6 Relations in the Lock Database

There are three ways in which the lock database can be accessed:

- As described in Section 4.2.2, the RSB for a given resource name can be located through the resource hash table. All locks associated with the resource can be located through the RSB state queue heads.
- As described in Section 4.2.5, the LKB for a given lock ID can be located through the lock ID table. The resource address field in the LKB points to the resource associated with the lock.
- All locks owned by a specific process can be located through the process lock queue.

Each process has a lock queue, a doubly linked list of all the locks it has requested. The listhead is in the PCB at the fields PCB$L_LOCKQFL and PCB$L_LOCKQBL. An LKB is linked into this list through the fields LKB$L_OWNQFL and LKB$L_OWNQBL. That is, PCB$L_LOCKQFL points not to the beginning of the first LKB in the queue but to field LKB$L_OWNQFL in that LKB. All granted locks are first, followed by converting and waiting locks. The locks are ordered this way to facilitate deadlock detection (see Section 4.4.2.2).

### 4.2.7 VMScluster Lock Database

Lock management data structures, RSBs and LKBs, are distributed among the nodes of a VMScluster system. This section provides an overview of how the lock management database is organized. In the following paragraphs,

wherever a node is said to perform an operation, the agent responsible for that operation is actually the distributed lock manager system application.

Each resource has a node responsible for granting locks on the resource, called the master node. The first node to define a root resource becomes the master node for that resource tree. The master node maintains a list of granted locks and a queue of waiting requests for the resource. Since the master node is responsible for granting a lock, a node must communicate a lock request to the master node as a first step in servicing the request.

Each node holding a lock on a resource mastered by another node has its own copy of the resource and lock blocks. The RSB on a node not mastering the resource contains the cluster system ID (CSID) of the mastering node in the field RSB$L_CSID. The RSB on the mastering node contains zero in that field to indicate that it is the master RSB. The CSID field also contains zero on a system that is not a VMScluster node.

In a VMScluster system, a process's lock request can be represented by two LKBs on two different nodes. Depending on whether a resource is mastered on the same node as a process requesting a lock on the resource, an LKB can be a local copy, a process copy, or a master copy:

- A local copy is an LKB for a lock request on one node whose resource is mastered on that same node. This LKB is the only one representing the process's lock. This is similar to the nonclustered case.
- A process copy is an LKB for a lock request on one node whose resource is mastered on another node. The process copy describes the process's interest in the resource. The other node has the master copy of the lock. The field LKB$L_REMLKID in the process copy identifies the lock ID of the master copy. (Lock IDs are specific to a single node.) RSB$L_CSID identifies the master node.
- A master copy is an LKB that exists on a node mastering a resource but that represents the lock of a process on a different node. The field LKB$L_REMLKID in the master copy identifies the lock ID of the process copy. The field LKB$L_CSID in the master copy identifies the node of the process copy.

The three types of LKB can be distinguished based on the setting of the bit LKB$V_MSTCPY in LKB$L_FLAGS and the contents of RSB$L_CSID in the associated resource's RSB:

- Local copy—LKB$V_MSTCPY is clear and RSB$L_CSID is zero.
- Process copy—LKB$V_MSTCPY is clear and RSB$L_CSID is nonzero.
- Master copy—LKB$V_MSTCPY is set and RSB$L_CSID is zero.

Each resource has a directory node responsible for mapping a resource's name to its master node. Given a resource name, the resource's directory node can be determined through a simple algorithm using the name string

and the number of directory nodes. A node can serve as the directory node for a subset of all root resources in a VMScluster.

When a lock is requested for a resource not locally defined, the local node first determines the directory node for the resource and sends a lock request message to it using system communication services (SCS). The directory node responds in one of the following ways:

- If the directory node finds the resource undefined, it sends a message to the requesting node to master the resource itself.
- If the directory node is the same as the master node, it performs the lock request and sends a confirmation message to the requesting node.
- If the directory node is not the same as the master node, it sends the identity of the master node to the requesting node.

The directory node for a particular resource maintains its directory entry RSB. A directory entry RSB has the RSB$V_DIRENTRY bit set in the RSB$L_STATUS field and the CSID of the resource's master node in the RSB$L_CSID field. If the directory node is also the resource's master node, the RSB$L_CSID field contains a zero, and one RSB serves both functions; if not, there are RSBs on both the directory and master nodes. Thus, there can potentially be an RSB on the directory node, an RSB on the master node, and an RSB on each node with a lock on the resource.

A node's relative participation in resource mastership is based on the value of its SYSGEN parameter LOCKDIRWT. The higher the value of LOCKDIRWT, the more a node participates in directory activity. In general, a LOCKDIRWT value of 0 is used for satellite nodes and a positive value for nodes with more processing power.

If a node that has mastered a resource tree is not the node on which most locking activity for that resource occurs, there is a potential performance loss because messages have to be exchanged between the master node and the nodes on which locking activity occurs. Mastership of a resource tree can move from one node to another during normal operation through dynamic remastering.

When the node mastering a resource tree receives a lock request from another node, it performs a quick test to determine whether the requesting node would be a more efficient master node:

- If the requesting node's LOCKDIRWT value is lower, then mastership does not need to change.
- If the requesting node's LOCKDIRWT value is the same, then if the amount of activity on the resource is greater than the system threshold and the requesting node has generated more of that activity, the requesting node will become the master.
- If the requesting node's LOCKDIRWT value is higher, it will become the master.

- If the current master no longer has an interest in the tree (if it has released all locks), the tree will move regardless of LOCKDIRWT.

If the resource should be remastered, the executive flags this and services the current request. At some later point, another code thread actually performs dynamic remastering.

### 4.2.8 Summarized Lock-Granting Behavior

A lock can be granted, converting, or waiting, depending on the modes of other locks on the resource. Table 4.1 lists the lock modes, their meanings, and the other granted lock modes with which each lock is compatible. This section provides a summary of the conditions under which locks are granted and describes the transitions between states. Further details can be found in Sections 4.3.2 and 4.3.3.

New lock requests are granted using the following basic rules:

- A lock on a new resource is granted immediately.
- A lock on an existing resource is granted if the conversion and wait queues are empty and its lock mode is compatible with those of locks already granted on the resource.
- A lock on an existing resource is granted immediately if it is a NL mode request and the LCK$V_EXPEDITE flag is specified.
- If none of the above criteria for granting a lock is met, the LKB is placed at the end of the wait queue. However, if the lock request was made with the LCK$V_NOQUEUE flag, the error status SS$_NOTQUEUED is returned and the LKB is not put on the wait queue.

Lock conversion requests are granted using the following rules:

- The conversion queue is ordered only by the arrival time of conversion requests at the resource master.
- A lock conversion to a less restrictive mode is granted synchronously because it is always compatible with the current set of granted locks.
- A lock conversion to a more restrictive mode is granted synchronously if it is compatible with the current set of granted locks. Note that because locks on the conversion queue remain granted in their original modes, they can block conversions. However, lock requests on the wait queue cannot block conversions.

  A lock cannot block its own conversion. Its request for conversion to a more elevated mode will not cause its own blocking AST to be queued.
- A request for a lock conversion to a more restrictive mode that specifies the LCK$V_QUECVT flag is placed at the end of the conversion queue even if it is compatible with the current set of granted locks, as long as the conversion queue is populated.

  Putting the request at the end of the queue means that it will not be granted until all lock requests queued ahead of it have been granted. This

simplifies some cases of application-locking protocol design by providing a mechanism to avoid indefinite postponement of queued conversions.

- If a request for a lock conversion to a more restrictive mode is not compatible with the current set of granted locks, the LKB is placed at the end of the conversion queue. The lock remains granted in its original mode. However, if the lock request was made with the LCKSV_NOQUEUE flag, the error status SS$_NOTQUEUED is returned and the LKB is returned to the granted queue in its original mode rather than placed on the conversion queue.
- When a granted lock is dequeued or converted to a less restrictive mode, the LKB at the head of the conversion queue is evaluated. If this request is compatible with the current set of granted locks, it is granted and the next conversion queue entry is evaluated. Evaluation ceases when the next conversion queue entry cannot be granted.

  If all conversion requests have been granted, and the conversion queue is empty, the LKB at the head of the wait queue is evaluated.

Waiting lock requests are granted using the following rules:

- The wait queue is ordered only by the arrival time of lock requests at the resource master.
- Lock requests on the wait queue are granted only when a granted lock is dequeued or converted to a less restrictive mode and no locks remain on the conversion queue.

## 4.3 LOCK MANAGEMENT SYSTEM SERVICES

The $ENQ system service attempts to grant a new lock or lock conversion request immediately. If the new lock or conversion request cannot be granted, the service places the LKB on the RSB's wait or conversion queue. The $DEQ system service dequeues or cancels a lock from a resource and then searches the resource's state queues for locks to grant that are compatible with the currently granted locks. The $GETLKI system service returns information about a specified lock or locks. The $SET_RESOURCE_DOMAIN system service associates a process with a resource domain, providing access to a wider name space for the process's resources.

Both synchronous and asynchronous forms are provided for the $ENQ and $GETLKI system services. The asynchronous forms, $ENQ and $GETLKI, validate the request and arguments, build appropriate data structures to represent the request, and return a status to the requestor that indicates whether the request was valid. The requestor can continue in its code stream or wait for the service to complete. The synchronous forms of these services, $ENQW and $GETLKIW, return control to the requestor only after the service is complete.

As a performance enhancement, a process can specify the FLAGS argument bit LCK$V_SYNCSTS on a $ENQ or $ENQW lock or conversion request; a request granted immediately causes the system service to return the status SS$_SYNCSTS and avoid the overhead of setting the event flag and delivering the AST. Chapter *System Service Dispatching* provides more information concerning synchronous and asynchronous system services. The following sections describe the operations of the lock management system services on a single node. VMScluster operation is mentioned, but the details are beyond the scope of this book.

### 4.3.1 The $ENQ[W] System Service

The $ENQ system service procedure, EXE$ENQ in module SYSENQDEQ, runs in kernel mode. EXE$ENQ first validates the event flag and lock mode arguments and tests accessibility of the lock status block. If any of these tests fails, EXE$ENQ returns to its requestor with an error status. If the tests succeed, EXE$ENQ tests whether LCK$V_CONVERT is set in the FLAGS argument to determine whether this is a new lock request or conversion of an existing lock. If the FLAGS argument indicates a lock conversion request, processing continues as described in Section 4.3.2.

Otherwise, for a new lock request, EXE$ENQ probes the resource name descriptor, returning the error status SS$_ACCVIO if the resource name is not accessible in the caller's mode. It validates the resource name length, returning the error status SS$_IVBUFLEN if the length exceeds 31. If the FLAGS argument bit LCK$V_EXPEDITE was specified, it tests that the requested lock mode was NL, returning the error status SS$_UNSUPPORTED if not.

When a new lock is requested, EXE$ENQ allocates an LKB and RSB from the caches or from nonpaged pool. It performs the cache access at IPL$_LCKMGR under the protection of a local cache interlock. EXE$ENQ allocates the RSB on the assumption that the resource is being defined for the first time.

At IPL 2, EXE$ENQ performs further argument validation and LKB initialization. It stores the process ID and the kernel thread ID in the LKB.

EXE$ENQ raises IPL to IPL$_LCKMGR and acquires the LCKMGR spinlock to synchronize access to the lock database. All exit and error paths release the LCKMGR spinlock and lower IPL before exiting. (In a VMScluster system, LCKMGR and SCS are synonyms for the same spinlock.)

If the requestor specified the PARID argument, EXE$ENQ performs the following:

1. EXE$ENQ verifies that the access mode of the $ENQ requestor is at least as privileged as that of the parent resource and not more privileged than that of the parent lock (for instance, a kernel mode lock cannot be a sublock of a user mode lock). It also verifies that the parent lock's PID matches that of the current process or, if not, that the lock is system-owned and the access mode of the $ENQ requestor is kernel or

executive. If any of these tests fails, EXE$ENQ returns the error status SS$_IVLOCKID to its requestor. If the tests complete successfully but the parent lock request has not been granted to this process, EXE$ENQ returns the error status SS$_PARNOTGRANT.

2. If the parent lock request has been granted to the current process, EXE$ENQ copies the parent lock's resource domain ID, in LKB$L_RSDM_ID, and the LKB$L_STS bits LKB$V_VALBLKRD and LKB$V_VALBLKWRT, which define access to the resource value block, into the new LKB.
3. EXE$ENQ verifies either that no value block is specified or that the caller has read access to it. No test for write access is necessary at this point, because the value block is only written on lock conversions or dequeues. EXE$ENQ returns the error status SS$_ILLRSDM on failure.
4. EXE$ENQ increments the reference count in the parent's lock and stores the parent lock's address in the new lock's LKB$L_PARENT field.
5. It copies the caller's access mode into LKB$B_RMOD. Because the LKB is also used as the ACB for AST delivery, the value in LKB$B_RMOD determines the access mode in which completion and blocking ASTs are delivered.
6. It copies the parent's RSB$B_RMOD, the access mode of the resource tree, and RSB$W_GROUP, the domain, into the new RSB. Note that EXE$ENQ ignores the ACMODE and RSDM_ID arguments if specified for a sublock.

Otherwise, when no parent resource exists, EXE$ENQ performs the following checks:

1. It validates the caller's default domain or the domain specified in the system service request, and verifies that its caller's access mode is at least as privileged as RDAB$L_ACMODE, the least privileged access mode allowed to access the domain.

   If a resource domain was specified in the $ENQ request, EXE$ENQ verifies that the resource exists and that the process has established an association with it, returning the error status SS$_NODOMAIN if not.

   If a resource domain was not specified in the $ENQ request, the normal case, EXE$ENQ locates the default process resource domain. If the process has not yet joined a default resource domain, EXE$ENQ establishes the association using the process's current UIC group. If the process's default resource domain does not match its current UIC group, EXE$ENQ changes its default domain to match.
2. It verifies that its caller's access mode is not more privileged than RDAB$L_DOM_ACMODE, the most privileged access mode allowed to access the domain. RDAB$L_DOM_ACMODE ensures that no locks survive the rundown of the process's domain association.

3. If the caller requested a systemwide resource name by specifying the FLAGS argument bit LCK$V_SYSTEM, EXE$ENQ checks that the process either has the SYSLCK privilege or requested the $ENQ system service from kernel or executive mode. If neither condition is true, EXE$ENQ returns the error status SS$_NOSYSLCK to its requestor.
4. If the caller's access mode is kernel or executive, no value block checks are required; privileged access modes are allowed to read and write the resource value block. Otherwise, EXE$ENQ copies the RSDM$V_READ and RSDM$V_WRITE bits, which determine read and write access to the resource value block, from RDAB$L_ACCESS into LKB$L_STS. It verifies that the caller has lock access to the domain, as determined by the bit RSDM$V_LOCK in RDAB$L_ACCESS, returning the error status SS$_ILLRSDM on failure.

   EXE$ENQ next verifies either that no value block is specified or that RSDM$V_READ is set. No test for write access is necessary at this point because the value block is only written on lock conversions or dequeues. It returns the error status SS$_ILLRSDM on failure.
5. If the ACMODE argument indicates a more privileged access mode than the caller's access mode, the lock is created instead in the caller's access mode. The chosen access mode is copied to LKB$B_RMOD and becomes the least privileged access mode from which the lock can be converted or dequeued, and the access mode in which completion and blocking ASTs are delivered.
6. The chosen access mode is also copied to RSB$B_RMOD, and becomes the access mode of the resource tree if the lock request specifies a new resource. Otherwise, if the resource is found to exist, this RSB will be deallocated to the RSB cache.
7. Finally, EXE$ENQ copies the group value from RDDB$L_GROUP into RSB$W_GROUP unless the resource is in the system domain.

EXE$ENQ copies the resource name and length as well as the parent, if any, into the RSB.

EXE$ENQ charges the lock against the job quota JIB$L_ENQCNT unless the request specified the FLAGS argument bit LCK$V_NOQUOTA, which requires that the request had been made from executive or kernel mode. (Use of this flag is reserved to Digital.) If this request would exceed the job's ENQLM quota, EXE$ENQ returns the error status SS$_EXENQLM. Otherwise, EXE$ENQ allocates a lock ID, expanding the lock ID table if necessary, and stores the address of the LKB in the table entry for that lock ID.

EXE$ENQ then determines whether the resource already exists on this node. It computes the resource hash value, indexes into the resource hash table, and searches the resource hash chain for the named RSB. For the resource to exist, the temporary RSB must match an RSB with the same values in the following fields: resource name, access mode, parent RSB address, and UIC group number (as determined by the resource domain).

If it does not find the RSB for the named resource, EXE$ENQ links the new RSB to the end of the hash chain. EXE$ENQ initializes the remaining RSB fields, including the three lock queue headers, the value block and sequence number, and the reference count.

If the resource has no parent, EXE$ENQ places it at the end of the systemwide list of root resources whose listhead is LCK$GL_RRSFL. The RSB's own address is stored in its root resource field, RSB$L_RTRSB. EXE$ENQ determines the CSID of the resource master and stores it, or zero if the local node is the master, in RSB$L_CSID.

If the resource has a parent, the new resource inherits its CSID from the parent resource. Resource depth is initialized to 1 more than the parent resource depth. If maximum lock depth is exceeded, EXE$ENQ puts the RSB and LKB into the caches and returns the error status SS$_EXDEPTH. The new resource also inherits the parent's root resource, RSB$L_RTRSB. It is inserted into the subresource queue of its parent and the parent's reference count is incremented.

If the resource is new and mastered locally, no further checks are necessary—the new lock is granted immediately (see Section 4.3.3).

If an RSB for the named resource is found, the new RSB is superfluous and is put into the RSB cache. If the resource is mastered locally, the new lock is granted immediately when the conversion and wait queues are empty and the request mode in the LKB is compatible with the currently granted locks (see Section 4.3.3). EXE$ENQ returns the success code SS$_SYNCH to its requestor if the FLAGS argument bit LCK$V_SYNCSTS is set. The event flag and completion AST are omitted in this case. Otherwise, EXE$ENQ returns the status SS$_NORMAL and proceeds to set the event flag and deliver the completion AST, if requested.

If an RSB is found but not mastered locally, the local node sends a lock request to the master node and waits the kernel thread requesting the lock for resource RSN$_SCS while cluster communication occurs on its behalf.

If the lock cannot be granted immediately, the FLAGS argument bit LCK$V_NOQUEUE determines EXE$ENQ's action:

- If LCK$V_NOQUEUE is set, EXE$ENQ puts the LKB into the LKB cache and returns the failure status SS$_NOTQUEUED to its requestor.
- If LCK$V_NOQUEUE is clear, EXE$ENQ sets the lock state to LKB$K_WAITING and places the LKB at the end of the wait queue in the RSB. The wait queue is maintained in first-in/first-out (FIFO) order. If the waiting LKB is not a master copy LKB, it is also queued onto PCB$L_LOCKQFL in the PCB of the requesting (current) process. If the LKB is not a process copy and has not disabled deadlock wait (LCK$V_NODLCKWT is clear), and if deadlock wait is enabled on the system (LCK$GL_WAITTIME is nonzero), then a due time is computed and the LKB is inserted into the timeout queue. Section 4.4.1 describes deadlock searches initiated by timeouts.

A null mode lock request on a resource that is currently granted in exclusive mode is a valid and compatible request. Such a request would normally be placed at the end of the resource's wait queue if another request is currently on the wait queue or conversion queue. If the LCK$V_EXPEDITE flag was specified in the $ENQ system service request, however, the request is granted immediately.

The asynchronous form of the system service ($ENQ) returns to its requestor. The synchronous form of the system service ($ENQW) waits both for the event flag associated with the request to be set and for status to be returned in the LKSB.

To speed checks for compatibility with the currently granted locks, each RSB contains a single field indicating the most restrictive granted lock mode of all locks in both the granted and conversion queues for that resource. This field is termed the group grant mode. Note that locks on the conversion queue retain their original grant mode while waiting for their conversion requests to complete. It is the original grant mode of these locks that is used in calculating the group grant mode, not their request mode.

The value of the group grant mode is stored in the RSB at the field RSB$B_GGMODE. Because this value is calculated when a lock is granted and maintained in the RSB, compatibility checking involves only one compare operation. Note that in a VMScluster system the group grant mode is maintained only in the master RSB.

### 4.3.2 Lock Conversions

When a process requests the $ENQ system service, the value of the LCK$V_CONVERT bit in the FLAGS argument differentiates between a new lock request and a lock conversion. When LCK$V_CONVERT is set, EXE$ENQ performs a lock conversion. EXE$ENQ obtains the lock ID of the lock to be converted from the LKSB argument and uses the LKMODE argument as the request mode.

Four lock modes affect EXE$ENQ's actions:

- The current mode of the converting lock, called its grant mode and stored in LKB$B_GRMODE
- The converting lock's desired new value, called its request mode and stored in LKB$B_RQMODE when the lock is on the conversion or wait queue
- The most restrictive grant mode found in a lock on the resource's conversion or granted queues, called the group grant mode and stored in RSB$B_GGMODE
- The blocking condition to compare against when locks are removed from the granted queue, called the conversion grant mode and stored in RSB$B_CGMODE

The conversion grant mode prevents a lock from blocking its own conversion and determines when an attempt to grant queued lock conversions is worthwhile. Most of the time, the conversion grant mode contains the same value as the group grant mode. The conversion grant mode differs from the group grant mode when both of the following are true:

- The grant mode of the lock at the head of the conversion queue is the most restrictive lock mode for the resource.
- No other locks are granted at that same lock mode.

In this case, the resource's conversion grant mode summarizes only the grant modes of locks on the granted queue. It contains a less restrictive lock mode than the group grant mode does, because group grant mode includes the grant modes of locks on the conversion queue.

EXE$ENQ begins by verifying that the lock exists, that its access mode allows access from the current mode, and that it is granted to the current process. Note that this check is made on the process ID rather than the kernel thread ID. EXE$ENQ then checks that the desired value block access, if any, is allowed, using the grant mode and the request mode to determine whether the value block will be read or written.

EXE$ENQ copies the current kernel thread ID into the LKB, stores the new completion and blocking AST addresses and AST parameter, if any, and saves the flags specified in the conversion request.

EXE$ENQ then removes the lock specified by the lock ID from the granted queue. If no locks remain on the granted or conversion queue, the converting lock is granted immediately and EXE$ENQ attempts to grant any waiting locks after clearing the group and conversion grant modes. Section 4.3.3 describes the grant procedure.

When additional locks exist on the conversion or grant queue, the conversion grant mode and the lock's grant mode are compared:

- If they are not equal, the compatibility of the converting lock's request mode and the resource's group grant mode determines whether the lock is granted or placed at the end of the conversion queue. Because the converting lock was not the most restrictive lock on the granted queue, its conversion has no effect on locks in the conversion or wait queue. EXE$ENQ will not attempt to grant any locks except the converting lock. If the FLAGS argument bit LCK$V_QUECVT was specified and the conversion queue is populated, EXE$ENQ places the converting lock at the end of the queue. Otherwise, EXE$ENQ grants the lock, as described in Section 4.3.3.
- If the lock's grant mode matches the resource's conversion grant mode, the converting lock was granted in the most restrictive lock mode present on the granted queue. The resource's group and conversion grant modes must be recalculated without including the grant mode of the converting lock, to prevent it from blocking its own conversion.

If the recalculated grant value proves compatible with the lock's request mode, the value is stored in the group grant and conversion grant fields. If the FLAGS argument bit LCK$V_QUECVT was specified and the conversion queue is populated, EXE$ENQ places the converting lock at the end of the queue. Otherwise, EXE$ENQ grants the lock, as described in Section 4.3.3. Since the change in this lock's status may be significant for other locks on the conversion or wait queue, EXE$ENQ attempts to grant locks first from the conversion queue, then from the wait queue, until it reaches a lock that it cannot grant.

In either case, if the lock's request mode is incompatible, EXE$ENQ tests the LCK$V_NOQUEUE bit:

- If LCK$V_NOQUEUE is set, EXE$ENQ puts the lock back on the granted queue and returns the status SS$_NOTQUEUED to the requestor.
- If LCK$V_NOQUEUE is clear, EXE$ENQ clears the lock state and places the LKB at the end of the FIFO conversion queue. EXE$ENQ does not alter the group grant mode but sets the conversion grant field to the recalculated value if the lock is first in the conversion queue. EXE$ENQ also moves the LKB to the end of the PCB queue. The PCB queue has granted locks first, followed by waiting and converting locks. If the LKB is not a process copy, if the conversion request did not disable deadlock wait, and if deadlock wait is enabled on the system (LCK$GL_WAITTIME is nonzero), then EXE$ENQ computes a due time and inserts the LKB into the timeout queue. Section 4.4.1 gives more information on deadlock searches initiated by timeouts.

Locks on the conversion or wait queue are granted later, by EXE$DEQ when blocking locks are removed from the granted or conversion queue, and by EXE$ENQ when blocking locks are converted to less restrictive lock modes.

Lock conversions, like lock grants, take place on the VMScluster node mastering the resource. If the resource is mastered on another node, the local node sends a lock conversion request to the master node and waits the kernel thread requesting the lock for resource RSN$_SCS while cluster communication occurs on its behalf.

### 4.3.3 Granting a Lock

The routine LCK$GRANT_LOCK, in module SYSENQDEQ, is called to grant a lock request. LCK$GRANT_LOCK is called under three different sets of circumstances:

- EXE$ENQ receives a request for a lock on a new resource or a resource with locks whose modes are compatible. The lock request can be granted immediately, synchronously with the original system service call.

- EXE$ENQ converts a lock on a resource to a less restrictive lock mode. Another lock that was blocked can now be granted, asynchronously to its original lock request.
- EXE$DEQ dequeues or cancels a lock on a resource. A lock that was blocked can now be granted, asynchronously to its original lock request.

LCK$GRANT_LOCK takes the following steps in granting a lock:

1. If the mode of the lock being granted is more restrictive than the existing group grant mode, LCK$GRANT_LOCK copies the mode of the lock being granted to the group grant mode field and the conversion grant mode field.
2. It places the LKB on the granted queue, changing its state to granted. LCK$GRANT_LOCK writes the requested lock mode in LKB$B_GRMODE.
3. If the lock is a master copy completing asynchronously, it invokes LCK$SND_GRANTED to send a granted message to the VMScluster node waiting for this lock.
4. If the lock is being granted asynchronously, it might be on the timeout queue. If so, LCK$GRANT_LOCK removes it.
5. After processing the AST delivery requirements described in this section, LCK$GRANT_LOCK invokes SCH$POSTEF to set the event flag associated with the lock request (LKB$B_EFN). If the kernel thread was waiting for this event flag to be set, its scheduling priority and state may be altered. Chapter 3 discusses event flags, and Chapter 2 gives information about scheduling.

The LKB contains the thread ID of the kernel thread that last requested the $ENQ system service for this lock. LCK$GRANT_LOCK delivers any ASTs associated with the lock request to this kernel thread. LCK$GRANT_LOCK makes a series of tests to determine whether an AST should be queued to the kernel thread whose lock request it granted. Three ASTs can be requested:

- A special kernel mode AST
- A user-requested blocking AST
- A user-requested completion AST

The three are independent of each other. Thus, it is possible that no AST will be requested or as many as three ASTs will be required.

LCK$GRANT_LOCK must queue a blocking AST to the kernel thread if it requested one and if the newly granted lock is blocking another lock. No blocking AST is necessary if none was requested or if the lock is not blocking another lock.

If the kernel thread requested a completion AST, LCK$GRANT_LOCK queues one unless the lock request was granted synchronously and the FLAGS argument bit LCK$V_SYNCSTS was set.

The special kernel mode AST must be queued if the lock request completed asynchronously. The special kernel mode AST routine writes the status to the process's LKSB and possibly a value to the resource value block. Even if the lock request completed synchronously, the special kernel mode AST routine is necessary to perform cleanup if a completion or blocking AST is to be queued.

An ACB can describe one normal AST procedure or one special kernel mode AST routine. An ACB can also describe a special kernel mode AST routine piggybacked on a normal AST procedure. Chapter *ASTs* gives a detailed description of ASTs. If an AST is required, LCK$GRANT_LOCK invokes SCH$QAST to queue an ACB to the process. The LKB is used as the ACB.

LCK$GRANT_LOCK chooses one of the following:

- It does not queue an ACB if the lock request is synchronous and neither a blocking nor a completion AST is required.
- It queues an ACB specifying a special kernel mode AST if the lock request is asynchronous and neither a blocking nor a completion AST is required.
- It queues an ACB specifying a piggyback special kernel mode AST if a blocking AST, a completion AST, or both are required.

Because the ACB can contain the procedure value of only one AST procedure, special treatment is required when both completion and blocking ASTs must be delivered. When the lock is granted, LCK$GRANT_LOCK writes the procedure value of the completion AST procedure (stored at the field LKB$PQ_CPLASTADR) in the field LKB$PQ_AST. It then queues the LKB as an ACB.

Just before entering the completion AST procedure, the AST delivery service routine dispatches to the piggyback special kernel mode AST routine. This routine writes the procedure value of the blocking AST (stored at the field LKB$PQ_BLKASTADR) in LKB$PQ_AST. It then requeues the LKB as an ACB. When the piggyback special kernel mode AST routine exits, the completion AST procedure executes. When the completion AST procedure exits, the blocking AST is delivered.

### 4.3.4 System-Owned Locks

Some locks, called system-owned locks, are not associated with any process. A system-owned lock, its resource, and thus its value block remain in existence when no process has any interest in the resource. A system-owned lock has zero in its LKB$L_PID field and is not queued to any PCB lock queue. The scope of its resource name may be systemwide or qualified by UIC group. Note the distinction between a system-owned lock and a resource that is defined systemwide.

A system-owned lock may only be requested from kernel or executive mode. The special SENQ system service FLAGS argument LCK$V_CVTSYS indicates that the lock should be granted as a system-owned lock or converted from a process-owned lock to a system-owned lock.

Although the service request must be made from kernel or executive mode, the access mode of the resource is determined by the $ENQ system service argument ACMODE, as it would be for any resource. One additional restriction applies—if a lock is system-owned, its parent lock (if any) must also be system-owned. A process-owned lock may have a system-owned lock as its parent, but a system-owned lock must not be a sublock of a process-owned lock.

The only possible state of a system-owned lock is granted. That is, a lock in a wait or conversion queue cannot be system-owned. This restriction exists partly because delivery of a completion AST or special kernel mode AST requires a process context. Furthermore, locks in the wait and conversion queues are examined during deadlock detection on the assumption that each lock is owned by a process.

When the FLAGS argument bit LCK$V_CVTSYS is set in a new lock request, EXE$ENQ sets the LCK$V_SYNCSTS and LCK$V_NOQUEUE flags as well. When LCK$V_NOQUEUE is set, EXE$ENQ returns the error status SS$_NOTQUEUED if it cannot grant the lock immediately. If it can grant the lock immediately and LCK$V_SYNCSTS is set, it does not queue a completion AST or set an event flag.

By specifying the FLAGS argument bit LCK$V_CVTSYS with LCK$V_CONVERT, a kernel thread can request the conversion of a process-owned lock to a system-owned lock or a system-owned lock to a less restrictive lock mode.

A kernel thread can request conversion of a system-owned lock to a more restrictive mode, but the request can succeed only if the conversion can complete immediately. Otherwise, the system-owned lock is converted to a process-owned lock, the lock remains granted at its original lock mode, and EXE$ENQ returns the error status SS$_BADPARAM.

A mechanism is defined for delivery of a blocking AST for a system-owned lock. The field LKB$PQ_BLKASTADR in a system-owned lock contains the procedure value of a blocking AST routine in system space. Instead of queuing a blocking AST to a process, the lock management services dispatch to that routine at IPL$_LCKMGR holding the LCKMGR spinlock.

Certain OpenVMS components, such as the Files-11 Extended QIO Processor (XQP), use system-owned locks. The XQP synchronizes access to the individual entries in its I/O buffer cache through system-owned locks. The XQP, running in the context of each process in the system, maintains a systemwide cache of blocks read from the on-disk file structure. A process's XQP requests a lock on a buffer cache entry only while it is reading or writing that entry in the cache. The cache entry exists, however, even when no

process is accessing it. The lock management data structures representing the cache entry must also continue to exist.

The use of system-owned locks is reserved to Digital. Any other use is strongly discouraged by Digital and completely unsupported.

### 4.3.5 Fork Process–Owned Locks

Lock management system services can be requested only from process context at IPL 0. Certain system components that execute in system context, such as the virtual I/O cache implementation, require access to locks also. The executive provides an internal interface to lock management functionality for fork processes. The use of this interface is strictly reserved to Digital; any other use is completely unsupported.

Locks created through this interface are called fork process–owned locks. The LKB$L_LCKCTX field of a fork process–owned lock points to an auxiliary structure called the lock context block (LCKCTX); for other types of lock, this field contains a zero. The LKB$L_PID field of a fork process–owned lock contains zero, just as that of a system-owned lock does. Deadlock detection is not possible for fork process–owned locks, just as it is not possible for system-owned locks. Unlike a system-owned lock, however, a fork process–owned lock can be waited or converted.

### 4.3.6 The $DEQ System Service

A kernel thread requests the $DEQ system service to dequeue locks or sublocks that are granted or to cancel ungranted lock requests. The $DEQ system service procedure, EXE$DEQ in module SYSENQDEQ, runs in kernel mode. EXE$DEQ examines the LKID argument and the FLAGS argument bit LCK$V_DEQALL to determine whether a specific lock or a number of locks are to be dequeued.

- If the FLAGS argument has the LCK$V_DEQALL bit set, then the kernel thread is requesting the dequeuing of multiple locks. The locks to be dequeued are determined by the $DEQ access mode argument ACMODE and by the LKID argument. The ACMODE argument is maximized with the access mode from which the $DEQ system service was requested.
  - If the LKID argument is specified, EXE$DEQ dequeues all sublocks of that lock whose access modes are not more privileged than the dequeue access mode. Note that if the LKID argument is specified with LCK$V_DEQALL, sublocks of that lock are dequeued, but the lock itself is not dequeued.
  - Otherwise, if the LKID argument is zero, EXE$DEQ checks every lock held by the process and dequeues each one whose lock access mode is not more privileged than the dequeue access mode.

- If the FLAGS argument has the LCK$V_DEQALL bit clear, then the kernel thread is requesting that one lock be dequeued or canceled. In this case, EXE$DEQ uses the LKID argument to locate the LKB and the FLAGS argument bit LCK$V_CANCEL to determine the operation.

To dequeue each individual lock, EXE$DEQ acquires the LCKMGR spinlock and raises IPL to IPL$_LCKMGR. Before dequeuing a PW or EX mode lock and writing the value block, EXE$DEQ verifies that process has write access to the value block. It verifies that the access mode of the $DEQ requestor is at least as privileged as that of the lock (LKB$B_RMOD) and that the lock PID matches that of the current process. If either of these tests fails, EXE$DEQ returns the error status SS$_IVLOCKID to its requestor. Once the lock is verified, EXE$DEQ checks whether the lock has sublocks. Before a lock is deleted, its sublocks must be dequeued. Unless the LCK$V_DEQALL flag is set, EXE$DEQ returns the error status SS$_SUBLOCKS.

All exit and error paths release the LCKMGR spinlock and lower IPL before exiting. When dequeuing multiple locks, EXE$DEQ releases and reacquires the spinlock between individual lock request cancellations.

EXE$DEQ removes the LKB from whichever resource queue it is found on.

- If the lock is dequeued from the granted queue, EXE$DEQ checks whether the LKB is the only lock on the resource. If so, EXE$DEQ removes the RSB from its resource hash chain and deallocates it. If other locks remain, EXE$DEQ recomputes the resource's group grant mode and conversion grant mode and attempts to grant locks on the conversion and wait queues.
- If the lock is dequeued from the conversion queue, it might have blocked other lock requests. If it was at the head of the queue, or if its grant mode is equal to the resource's conversion grant mode, EXE$DEQ recomputes the resource's group grant mode and conversion grant mode. EXE$DEQ attempts to grant locks beginning with the new first lock in the conversion queue. It repeats this with the conversion and wait queues until it reaches a lock whose lock mode is incompatible with the resource group grant mode.
- If the lock is dequeued from the head of the wait queue and the conversion queue is empty, EXE$DEQ tries to grant the first lock in the wait queue. If it succeeds, EXE$DEQ continues with the next lock in the wait queue. It repeats this until it reaches a lock whose lock mode is incompatible with the resource group grant mode.

If the lock being dequeued was a sublock, EXE$DEQ decrements its parent lock's reference count. It releases the lock ID and removes the LKB from the process's PCB lock queue.

If the lock was waiting or in the conversion queue, EXE$DEQ sets the event flag associated with the lock request and queues the LKB as an ACB to the kernel thread to return final lock status. The LKB is deallocated when the AST is delivered.

If the lock was granted, its LKB may still be queued as an ACB. If the ACB was merely to deliver a blocking AST, EXE$DEQ removes the LKB/ACB from the ACB queue and deallocates the LKB. Otherwise, the LKB/ACB will be deallocated when the AST is delivered. Whenever the LKB is deallocated, the lock quota is returned to the process.

### 4.3.7 The $GETLKI[W] System Service

The $GETLKI[W] system service enables a process to obtain information about one or more locks that it is allowed to interrogate. The process may only obtain information about locks on resources on the same node in a VMScluster system with access modes equal to or less privileged than the access mode at which the $GETLKI request is issued. For example, a process running in user mode cannot obtain information about locks on executive mode resources. The field RSB$B_RMOD defines the resource access mode.

The process can be further limited to a subset of the resource name space by its lack of privilege. Without any privilege, a process can only interrogate locks on resources within its default process resource domain (its UIC group) and in resource domains that it has joined using the $SET_RESOURCE_DOMAIN system service. With WORLD privilege, a process can interrogate locks on resources of any resource domain. Obtaining information about the locks of systemwide resources requires either that the process have SYSLCK privilege or that it make the $GETLKI request from kernel or executive mode.

The $GETLKI system service procedure, EXE$GETLKI in module SYSGETLKI, runs in kernel mode. The system service is called with a LKSB argument that either identifies a particular lock or specifies a wildcard operation. First, EXE$GETLKI locates the LKB associated with the specified lock ID and verifies that the process can interrogate it. If the process specified a wildcard operation, EXE$GETLKI locates the first LKB that the process can interrogate. EXE$GETLKI begins with lock index 0 and scans the lock ID table. On each successive call, it returns information about one lock, maintaining the lock index context for the next call.

EXE$GETLKI is called with the address of an item list that includes, for each specified item, which kind of lock information is to be returned, the size and address of the buffer to receive the information, and a location to receive the size of the information returned. EXE$GETLKI checks each item in the item list for correctness: its item code must be valid; its buffer descriptor and buffer must be writable in the access mode of $GETLKI's requestor.

In general, it then copies the requested information, either from the LKB or its RSB, to the buffer and records the size of the returned information in the specified location.

Certain types of information are not obtainable through simply copying data structure fields, for example, a list of all locks blocking the specified lock. EXE$GETLKI contains special routines for getting such information.

When EXE$GETLKI has either processed all items in the item list or found one that is incorrect or that has an inaccessible buffer, it is done. It sets the event flag associated with the request and queues a completion AST if one was requested and if the system service completed without error. EXE$GETLKI then returns to its requestor with completion status in R0.

### 4.3.8 The $SET_RESOURCE_DOMAIN System Service

The $SET_RESOURCE_DOMAIN system service manages a process's association with a resource domain, a name space determining the processes that can share access to locks and their value blocks.

The $SET_RESOURCE_DOMAIN system service procedure, EXE$SET_RESOURCE_DOMAIN in module RESOURCE_DOMAIN_OBJECT, runs in kernel mode. It associates the process with the desired resource domain, creating data structures to represent the domain if they do not already exist. The AUDIT_SERVER process ensures that the data structures correctly represent the object security profile for the resource domain. Figure 4.4 shows the layout of these data structures, the RDDB, the RDPB, and the RDAB.

EXE$SET_RESOURCE_DOMAIN's actions are determined by the function code argument, which specifies whether the process wishes to join or leave a domain.

If the process is joining a domain, the domain number argument identifies the desired domain; it must be a legal UIC group number, a value between 1 and $3FFE_{16}$. (Note that the Authorize utility expresses UIC group numbers in octal). EXE$SET_RESOURCE_DOMAIN returns the error status SS$_BADPARAM if this argument is invalid.

EXE$SET_RESOURCE_DOMAIN returns the error status SS$_ACCVIO if the user does not have write access to the RSDM_ID argument.

EXE$SET_RESOURCE_DOMAIN locates the RDDB for the desired domain, creating it if necessary. It calls EXE$CHKPRO_INT, an internal interface to the Check Protection ($CHKPRO) system service, with the associated ORB to determine the process's access rights to the resource domain. EXE$SET_RESOURCE_DOMAIN returns the error status SS$_NOPRIV if the access check fails. Otherwise, it increments the RDDB reference count under the protection of the RSDM mutex, EXE$GL_RSDM_MUTEX, and sets the RDAB to reflect the RDDB and the allowed access. It uses the RDAB$L_SEQ field and the index of the RDAB within the RDPB to create the RSDM_ID, which it returns to its caller.

If the process is leaving the domain, EXE$SET_RESOURCE_DOMAIN ignores the domain number argument—the RSDM_ID argument must locate a valid RDAB with matching sequence number. If the user does not have read access to the RSDM_ID argument, EXE$SET_RESOURCE_DOMAIN returns the error status SS$_ACCVIO. Otherwise, it tests that the process did not specify the system domain or the default process domain; attempting to leave these domains produces the errors SS$_UNSUPPORTED and SS$_RSDMNOTFOU, respectively.

A process cannot leave a domain in which it still holds locks. EXE$SET_RESOURCE_DOMAIN uses the RSDM_ID argument to locate the RDAB and RDDB, verifies that no locks are held for the domain, decrements the RDDB reference count under the protection of the RSDM mutex, and clears the process's RDAB fields, breaking the process's association with the domain.

## 4.4 HANDLING DEADLOCKS

A deadlock occurs when several locks are waiting for each other in a circular fashion. The executive resolves deadlocks by choosing a participant in the deadlock cycle and refusing that participant's lock request. The participant chosen to break the deadlock is termed the victim. The victim's lock or conversion request fails, and the error status SS$_DEADLOCK is returned in the victim's lock status block.

None of the victim's already granted locks are affected, even when they are part of the deadlock. Resolution of the deadlock is the responsibility of the victim.

There are three phases of deadlock handling:

1. A deadlock is suspected.
2. A deadlock search proves that a deadlock actually exists.
3. A victim is chosen.

These three phases are described in subsequent sections. The descriptions are limited to handling of deadlocks within one system that is not a VMScluster node. VMScluster deadlock handling is beyond the scope of this book.

### 4.4.1 Initiating a Deadlock Search

Because deadlock detection is time-consuming, it is not desirable to search for deadlocks every time a lock or conversion request is blocked. Instead, the executive searches for a deadlock only when a lock request has been waiting for a resource for a specified amount of time. The SYSGEN parameter DEADLOCK_WAIT specifies how many seconds a blocked lock request must have been waiting before a deadlock search is initiated.

A way of restricting a particular lock's participation in deadlock searches is provided through the special $ENQ FLAGS arguments LCK$V_NODLCKWT and LCK$V_NODLCKBLK. The LCK$V_NODLCKWT flag in a lock or conversion request inhibits the deadlock search mechanism on a per-lock basis. Locks requested in this manner cannot initiate conversion deadlock searches because they never time out. They are disregarded in multiple resource deadlock searches initiated for other locks. Incorrect use of this flag may cause genuine deadlocks to be ignored, however. For more information, consult the *OpenVMS System Services Reference Manual.*

When a lock request specifies a blocking AST procedure that dequeues the blocking lock or converts it to a less restrictive mode, that lock request may also specify the LCK$V_NODLCKBLK flag. This exempts the LKB from multiple resource deadlock searches on the assumption that the potential deadlock condition will be resolved by the blocking AST procedure. Incorrect use of this flag may cause genuine deadlocks to be ignored. For more information, consult the *OpenVMS System Services Reference Manual.*

When an LKB requested without the flag LCK$V_NODLCKWT is placed into a conversion or wait queue, EXESENQ also places the LKB into the lock timeout queue. The lock timeout queue listhead is at global location LCK$GL_TIMOUTQ. The AST queue fields in the LKB link it into the lock timeout queue. Figure 4.8 shows LKBs on the timeout queue.

When an LKB is placed into the timeout queue, the time at which the lock request will time out is computed and stored in LKB$L_DUETIME. (LKB$L_DUETIME is actually a double use of the special kernel mode AST procedure value field, LKB$L_KAST.) The due time is the sum of DEADLOCK_WAIT, stored in LCK$GL_WAITTIME, and the current system uptime in seconds, EXE$GL_ABSTIM.

Once every second, the routine EXE$TIMEOUT, in module TIMESCHDL, executes. EXE$TIMEOUT has various functions (see Chapter *Time Support*). One of them is to check whether the first entry in the lock timeout queue has timed out by comparing its LKB$L_DUETIME to the contents of EXE$GL_ABSTIM. Because the queue is time-ordered, checking the due

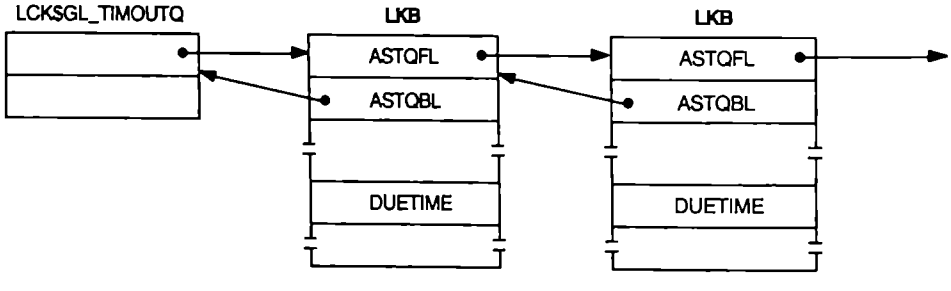



**Figure 4.8**
Lock Timeout Queue Ordered by LKB$L_DUETIME

time of the first entry is sufficient to determine whether a deadlock search is necessary. If the first entry has not timed out, no other entry could have. If the first entry has timed out, EXE$TIMEOUT initiates a deadlock search by invoking the routine LCK$SEARCHDLCK, in module DEADLOCK.

### 4.4.2 Deadlock Detection

There are two forms of deadlock, each requiring a different detection method. A conversion deadlock is easily detected because it is restricted to locks for a single resource. A multiple resource deadlock is harder to detect, requiring a more complex search.

#### 4.4.2.1 Conversion Deadlocks.

A conversion deadlock can occur when there are at least two LKBs in an RSB's conversion queue for a resource. If the request mode of one lock in the queue is incompatible with the grant mode of another lock in the queue, a deadlock exists.

For example, assume there are two protected read (PR) mode locks on a resource. The process with one PR mode lock requests a conversion to EX mode. Because PR mode is incompatible with EX mode, the conversion request must wait. While the first conversion request is waiting, the process with the second PR mode lock also requests a conversion to EX mode. The first lock cannot be granted because its request mode, EX, is incompatible with the second lock's grant mode, PR. The second conversion request cannot be granted because it is waiting behind the first.

The search for a conversion deadlock begins with the first LKB on the lock timeout queue. The LKB's state queue backward link points to the previous LKB in the conversion queue. The grant mode of the previous lock is compared with the request mode of the lock that timed out. If the modes are compatible, the next previous lock in the conversion queue is examined. The test is repeated until an incompatible lock is found or the beginning of the queue is reached. The flags LCK$V_NODLCKWT and LCK$V_NODLCKBLK are ignored.

If a lock with an incompatible grant mode is found, a deadlock exists. A victim LKB is selected (see Section 4.4.3). If the beginning of the queue is reached, a conversion deadlock does not exist, and a search for a multiple resource deadlock is initiated.

#### 4.4.2.2 Multiple Resource Deadlocks.

A multiple resource deadlock occurs when a circular list of processes are each waiting for one another on two or more resources.

For example, assume process A locks resource 1 and process B locks resource 2. Process A then requests a lock on resource 2 that is incompatible with B's lock on resource 2, and thus process A must wait. Note that at this point, a circular list does not exist. When process B then requests a lock on

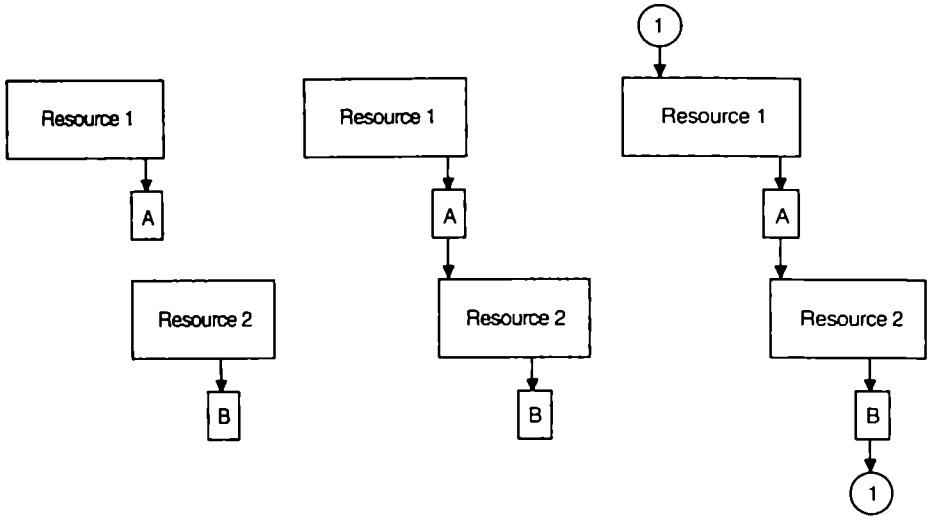



**Figure 4.9**
Example of a Deadlock

resource 1 that is incompatible with A's lock on resource 1, it must wait. A multiple resource deadlock now exists. Processes A and B are both waiting for each other to release different resources. These steps are shown in Figure 4.9. In the figure, locks that are blocking a resource (incompatible with waiting locks) are shown beneath the resource; locks that are waiting for a resource are shown above the resource.

This type of deadlock normally involves two or more resources, unless one process locks the same resource twice. (Usually a process does not lock the same resource twice. However, if the process is multithreaded, double locking can occur. Double locking can result in a multiple resource deadlock.)

To verify that a multiple resource deadlock exists, LCK$SEARCHDLCK uses a recursive algorithm. Its approach is based upon the following:

- A waiting lock is blocked by locks owned by other processes.
- Any of the other processes might themselves have waiting locks.
- Those waiting locks are blocked by locks owned by other blocking processes.

LCK$SEARCHDLCK starts with the lock that timed out on the lock timeout queue. It saves the extended process ID (EPID) of the owner process of the lock that timed out and invokes the multiple resource deadlock routine (LCK$SRCH_RESDLCK). If it finds a lock with the same owner EPID blocking a resource, a deadlock exists.

Each time LCK$SRCH_RESDLCK is called, a stack frame is pushed onto the stack. Each stack frame contains information on the current position in the search. Figure 4.10 shows the contents of the stack frame.

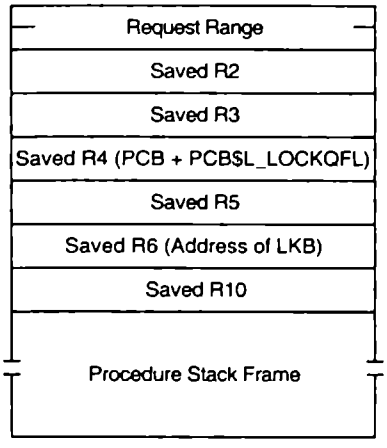



**Figure 4.10**
Stack Frame Built for LCK$SRCH_RESDLCK

The deadlock search algorithm is recursive. SYSGEN parameters KSTACKPAGES and DLCKEXTRASTK limit the number of kernel stack bytes that can be used for deadlock searches, and hence the amount of recursion. KSTACKPAGES is the size of the kernel stack, and DLCKEXTRASTK is the number of kernel stack bytes that cannot be used for deadlock searches. The difference between the available kernel stack and DLCKEXTRASTK is the amount of stack space available for LCK$SRCH_RESDLCK's stack frames. The reserved stack space allows interrupt service routines, which execute on the current process's kernel stack, sufficient stack space for execution.

Each invocation of LCK$SRCH_RESDLCK specifies the address of a waiting LKB. The resource associated with the LKB is located, and the resource state queues are searched for LKBs whose granted or requested lock mode is incompatible with that of the waiting LKB. If an incompatible LKB is found, that lock is considered to be blocking the waiting LKB unless it has the LCK$V_NODLCKBLK bit set in the LKB$L_FLAGS field.

When a blocking lock is found, its EPID is compared to that of the lock that initiated the deadlock search:

- If they are the same, the list is proved to be circular and a deadlock exists. A victim lock is chosen (see Section 4.4.3), and deadlock detection returns control to EXE$TIMEOUT.
- If the EPID of the blocking lock is not the same as the saved EPID and the search bitmap does not indicate that this process has been visited already, the PCB lock queue of the process owning the blocking lock is searched. If an LKB is found in a convert or wait state with the LCK$V_NODLCKWT bit clear, another invocation of LCK$SRCH_RESDLCK is made, specifying that LKB's address.

Each time LCK$SRCH_RESDLCK is called, it searches the state queues associated with the specified LKB to see if it is waiting for a resource.

When all the state queues for a given resource have been searched and no blocking lock has been found for that LKB, the routine removes the stack frame and returns control to its caller. If the caller itself was LCK$SRCH_RESDLCK, the previous search for blocked locks on the resource can now be resumed.

A process bitmap is maintained to reduce the number of repeated searches for blocking locks on a particular process. Each time a new blocking PCB is located, a bit corresponding to that process is set. If the bit for the PCB is set already, the search for locks blocking that process is terminated because its locks have been searched already.

4.4.2.3 **Unsuspected Deadlocks.** Note that the use of the process bitmap speeds the location of the suspected deadlock but prevents the accidental detection of unsuspected deadlocks. An unsuspected deadlock is one that exists within the lock management database but that has not been detected so far because none of its locks has timed out on the lock timeout queue. This behavior is accepted for the following reasons:

- Applications are assumed to be designed so that deadlocks are rare.
- Finding a process a second time in a deadlock search does not necessarily indicate that an unsuspected deadlock exists.
- The occurrence of unsuspected deadlocks should be rarer still.
- Any deadlock search that does not find a deadlock wastes processor time.
- The unsuspected deadlock will become a suspected deadlock when one of its own locks times out on the lock timeout queue and a deadlock search is initiated on its behalf.

Figure 4.11 shows two deadlocks. In the figure, locks that are blocking a resource (incompatible with waiting locks) are shown beneath the RSB; locks that are waiting for a resource are shown above the RSB. One deadlock is suspected and a search is in progress for it. The heavy arrows in the figure show the path of that deadlock cycle. The other is unsuspected. This figure is an extension of the deadlock cycle shown in Figure 4.9.

In this case, the deadlock search was initiated as a search for the locks blocking process A. Because process C's lock is the first one found granted for resource 2, it is the first lock that is investigated for participation in the deadlock cycle. Process C is waiting for resource 3. The bit corresponding to process C is set in the process bitmap. The context of the search is saved on the stack, and LCK$SRCH_RESDLCK is called to search for processes blocking process C's lock.

Process D has a blocking lock on resource 3. Process D is also waiting for resource 2. The bit corresponding to process D is set in the process bitmap. The context of the search is saved on the stack and LCK$SRCH_RESDLCK

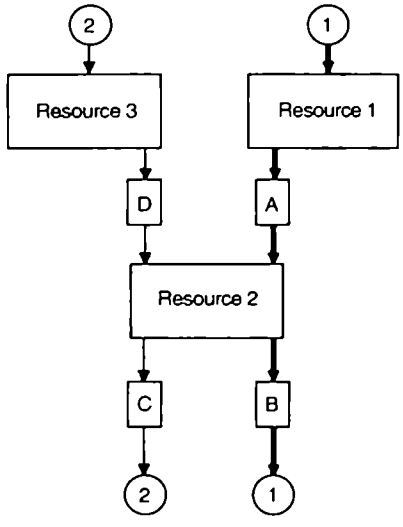



**Figure 4.11**
Suspected and Unsuspected Deadlocks

is called to search for processes blocking process D's lock. Process C has a blocking lock on resource 2. This situation is a deadlock. However, because the bit corresponding to process C was set in the process bitmap, the deadlock search for process C is abandoned. One by one, the stack frames are removed and the search whose context was saved continues. Eventually the deadlock search continues with locks blocking resource 2, and the deadlock cycle of processes A and B is discovered.

Eventually one of the locks requested by processes C and D will time out, and a deadlock search will be initiated.

4.4.2.4 **Example of a Search for a Multiple Resource Deadlock.** Figure 4.12 shows a series of locks that result in a deadlock. In the figure, locks that are blocking a resource (incompatible with waiting locks) are shown beneath the RSB; locks that are waiting for a resource are shown above the RSB. The heavy arrows in the figure show the path of the deadlock cycle.

Assume that the lock owned by process A timed out. Process A is waiting for a lock on resource 1. The deadlock search routine saves process A's EPID and invokes LCK$SRCH_RESDLCK, passing the address of process A's LKB.

The first incompatible lock on resource 1 is owned by process C. Process C has no other waiting locks, so LCK$SRCH_RESDLCK moves on to the next incompatible lock. This lock is owned by process D. When LCK$SRCH_RESDLCK follows the PCB queue for process D, it finds that this process is waiting for a lock on resource 3.

LCK$SRCH_RESDLCK invokes itself, passing the address of the LKB owned by process D. The new invocation of LCK$SRCH_RESDLCK pushes a stack frame detailing the position of the search on resource 1, and

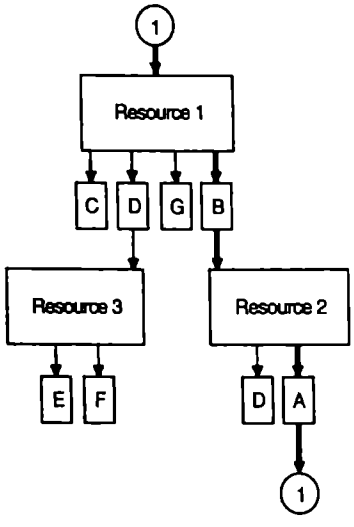



**Figure 4.12**
Example of a Multiple Resource Deadlock

LCK$SRCH_RESDLCK starts to search for locks on resource 3 that are incompatible with process D's lock. Resource 3 has two incompatible locks, owned by processes E and F. Neither of these processes is waiting for a lock, so the search on resource 3 terminates. The contents of the stack frame are restored and LCK$SRCH_RESDLCK returns to its previous invocation. The search for processes blocking process A resumes.

The next incompatible lock found on resource 1 is owned by process G. Process G has no waiting locks, so the search continues with process B. The PCB queue for process B shows that it is waiting for a lock on resource 2.

Again, LCK$SRCH_RESDLCK invokes itself, passing the address of the LKB owned by process B. The new invocation of LCK$SRCH_RESDLCK pushes a new stack frame onto the stack, and LCK$SRCH_RESDLCK finds that process D owns a lock that is incompatible with the lock owned by process B. However, because locks owned by process D have been searched already (the bit for process D is set in the process bitmap), the search moves on to the next process.

The next incompatible lock is owned by process A. Because the EPID of process A matches the EPID that was saved initially, the list is proved to be circular and a deadlock exists. Now a victim must be chosen.

### 4.4.3 Victim Selection

Because conversion deadlocks involve only two processes, the victim selection routine simply chooses the process with the lower deadlock priority, stored in the PCB at the field PCB$L_DLCKPRI.

For a multiple resource deadlock, the victim selection routine is only slightly more complicated. The frames that were pushed onto the stack in

each recursion into the deadlock location routine are searched for the lowest deadlock priority. Each time a lower deadlock priority value is found, the priority and the owner process are noted. If a deadlock priority of zero is found, that process is immediately chosen as the victim. When all frames have been searched or a deadlock priority of zero is found, the stack pointer is restored and the process with the lowest deadlock priority is chosen as the victim.

Note that the current OpenVMS implementation initializes the deadlock priority of all new processes to zero. Thus, it is not possible to determine which process will be chosen as the victim. With the current implementation, victim selection depends primarily on timing.

## 4.5 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIB]LKBDEF.SDL
[LIB]RDABDEF.SDL
[LIB]RDDBDEF.SDL
[LIB]RDPBDEF.SDL
[LIB]RSBDEF.SDL
[SYS]DEADLOCK.MAR
[SYS]INIT.MAR
[SYS]LCKMGR_MACROS.MAR
[SYS]RESOURCE_DOMAIN_OBJECT.B32
[SYS]SYSENQDEQ.MAR
[SYS]SYSGETLKI.MAR
[SYS]TIMESCHDL.MAR
[SYSLOA]DSTRLOCK.MAR

# 5 Kernel Threads

A noiseless, patient spider,
I mark'd, where, on a little promontory, it stood, isolated;
Mark'd how, to explore the vacant, vast surrounding,
It launch'd forth filament, filament, filament, out of itself;
Ever unreeling them—ever tirelessly speeding them.
Walt Whitman, *A Noiseless, Patient Spider*

Previous OpenVMS versions provided only a single execution context for a process and thus limited support for multithreading within a process. OpenVMS Alpha now supports multiple execution contexts, enabling more parallelism for an application's threads, which can execute concurrently on multiple CPUs.

This chapter describes the implementation of that support and the interaction between the OpenVMS executive and a user mode thread manager, in particular, the DECthreads thread manager.

## 5.1 OVERVIEW

A thread of execution is a sequential flow of instruction execution. A traditional program is single-threaded, consisting of one main thread of execution. In contrast, a multithreaded program divides its work among multiple threads that can run concurrently. OpenVMS supplies DECthreads, a runtime library, to facilitate the creation, synchronization, and deletion of multiple user mode threads.

OpenVMS Alpha Version 7.0 supports multiple execution contexts within a process: each execution context has its own hardware context and stacks, and can execute independently of the others; all execution contexts within a process share the same virtual address space. The term *kernel thread* refers to one of these execution contexts and the thread of execution running within it. This volume uses the term *multithreaded* to refer to a process with multiple kernel threads.

The kernel thread is now the basis for OpenVMS scheduling. Section 5.3 documents the executive data structures that describe kernel threads. This new capability enables an application to create additional execution contexts and specify the code to be executed within them. Sections 5.4 and 5.5 describe many of the system services related to this capability.

When code executing in one kernel thread incurs a page fault or enters some other kind of wait, the process's other kernel threads are unaffected and can continue execution. Each kernel thread has its own quantum. Switching context from one kernel thread to another, whether or not they are from

different processes, requires a full hardware context switch, basically the same steps that make up process context switch in prior OpenVMS versions. This model of execution scheduling is sometimes referred to as one-to-one threading.

Adding this capability required fundamental changes in scheduling and process models and intraprocess synchronization. It required some changes in the handling of the process working set list. Section 5.6 summarizes the changes in the OpenVMS scheduling model. Section 5.7 discusses some of the implications of multiple kernel threads' sharing an address space, and Section 5.8 describes the executive's intraprocess synchronization techniques.

The term *kernel thread* is a somewhat confusing name, in that instructions executing in a kernel thread can run in any of the four access modes, not only kernel mode. The term arose to distinguish the current multithreading model, supported within the OpenVMS executive (or kernel), from the previous model, supported only by DECthreads.

The DECthreads run-time library contains procedures to create and coordinate threads. Its threads and its thread manager run only in user mode. Its thread manager can multiplex user mode threads in a single kernel thread with low-overhead context switching.

With no executive support, previous versions of DECthreads could only provide timer-based switching of user mode threads within one process on one CPU. If a thread incurred a page fault or went into a wait state, all the threads in the process were stalled. Even in a symmetric multiprocessing (SMP) system, only one thread could execute at a time. A major goal for executive multithreading support was to address these restrictions.

With executive multithreading support, when a kernel thread executing in user mode is about to be waited, the OpenVMS executive notifies DECthreads, so that the thread manager has the opportunity to switch a different, runnable user mode thread into the context of that kernel thread, thus preventing the kernel thread's being waited. This notification is referred to as an upcall because an inner mode component must trigger a call to an outer mode component. The executive also makes upcalls to notify DECthreads of events that unblock user mode threads, which may then be switched back to execute in any available kernel thread. Because of upcalls, a DECthreads application has increased parallelism, even if running in a process with a single kernel thread.

The DECthreads model of scheduling user mode threads within an OpenVMS process is referred to as two-level scheduling. In a process with a single kernel thread, DECthreads performs many-to-one threading. Running in a multithreaded process (one with multiple kernel threads), DECthreads performs many-to-few threading. Section 5.9 provides more information.

Although kernel thread support is intended largely as a feature to aid user mode multithreading, this chapter's focus is on the executive code and data

structures that implement that support. Section 5.10 provides an overview of the DECthreads user mode thread manager.

## 5.2 MULTITHREADING CONTROLS

The following parameters affect the degree to which multithreading is possible systemwide and in a particular process:

- The SYSGEN parameter MULTITHREAD
- The DECthreads logical name PTHREAD_CONFIG

The SYSGEN parameter MULTITHREAD controls the maximum number of kernel threads any process on the system can create. A value of 0 means each process is limited to its initial kernel thread, and no upcalls are performed in any process. A value of 1 limits each process to its initial kernel thread but allows upcalls to a user mode thread manager whenever appropriate. The maximum value of the parameter is 16.

A process can define the logical name PTHREAD_CONFIG to affect the operation of the DECthreads user mode thread manager within that process. The logical name is defined as a string of semicolon-separated options. The option called vp-count controls the maximum number of kernel threads DECthreads tries to create in a particular process on a system with a MULTITHREAD value larger than 1. If this option is absent or if the logical name is not defined, the default DECthreads maximum is the number of CPUs currently in the system.

The possible scenarios for multithreading are as follows:

- MULTITHREAD value of 0—A system can be set up to run with no more multithreading capabilities than existed in previous releases. That is, each process has only one kernel thread. If the DECthreads thread manager is running in the process, it can schedule user mode threads within the kernel thread, but *no* upcalls are made to the thread manager.
- MULTITHREAD value of 1—Each process has only one kernel thread. In a process running with the DECthreads thread manager, upcalls are made whenever appropriate.
- MULTITHREAD value larger than 1—A process may create multiple kernel threads, and if the DECthreads thread manager is running, upcalls are made whenever appropriate.

## 5.3 KERNEL THREAD DATA STRUCTURES

Support for multithreaded processes requires separating processwide information from thread-specific data and replicating thread-specific data.

Some process control block (PCB) information has been moved to a nonpaged pool structure called a kernel thread block (KTB) (see Section 5.3.2). A process has one PCB and can have up to 16 KTBs.

Some process header (PHD) information is replicated in a floating-point register and execution data structure (FRED) (see Section 5.3.3).

Each thread has a private P1 space area, which includes space for the thread's four stacks and which replicates some P1 data cells, in a data structure called the per kernel thread area (PKTA) (see Section 5.3.4).

Figure 5.1 shows some of the relations among these data structures. For simplicity, it omits some of the pointers between structures. Each of the blocks labeled User Mode Thread represents a DECthreads thread-specific data structure and a user mode stack. Section 5.10.2 provides further details.

The sections that follow describe how individual threads are identified and discuss thread-specific data structures in more detail.

### 5.3.1 Kernel Thread Identification

A kernel thread within a process is identified by its thread ID number, a number from 0 to 15. The initial kernel thread is always thread ID 0. When a new kernel thread is created in a process, it gets the next available thread ID.

A longword internal process ID (IPID) consists of two words: the low-order word is a process index number, and the high-order word is a sequence number identifying the usage of that index number. When a process is created, it is assigned an index number not currently in use. The sequence number associated with the index number is concatenated to form the IPID. The IPID is transformed into an extended process ID (EPID) that uniquely identifies the process within the entire VMScluster.

The process's IPID also identifies the initial kernel thread in the process and is recorded in CTL$GL_INITIAL_PID.

Each additional kernel thread in a process is also uniquely identified within the whole system by its own IPID and within the entire VMScluster by its EPID. These IDs are based upon the process's IDs. When an additional kernel thread is created, its IPID is formed by adding its thread ID to the sequence number in the initial kernel thread's IPID. In other words, a range of sequence numbers is associated with the process index used by a multithreaded process.

Within a multithreaded process, IPIDs can be reused as kernel threads are created and deleted. In particular, if a process reruns a multithreaded application, the IPIDs of the created kernel threads are the same from run to run.

Chapter 7 has further details on IPIDs and EPIDs. Section 5.10.3 discusses how user mode threads are identified.

### 5.3.2 Kernel Thread Block (KTB)

A KTB exists for each kernel thread the process creates.

To minimize changes to the PCB between OpenVMS Alpha Version 7.0 and previous releases, the KTB for the initial (and perhaps only) kernel thread

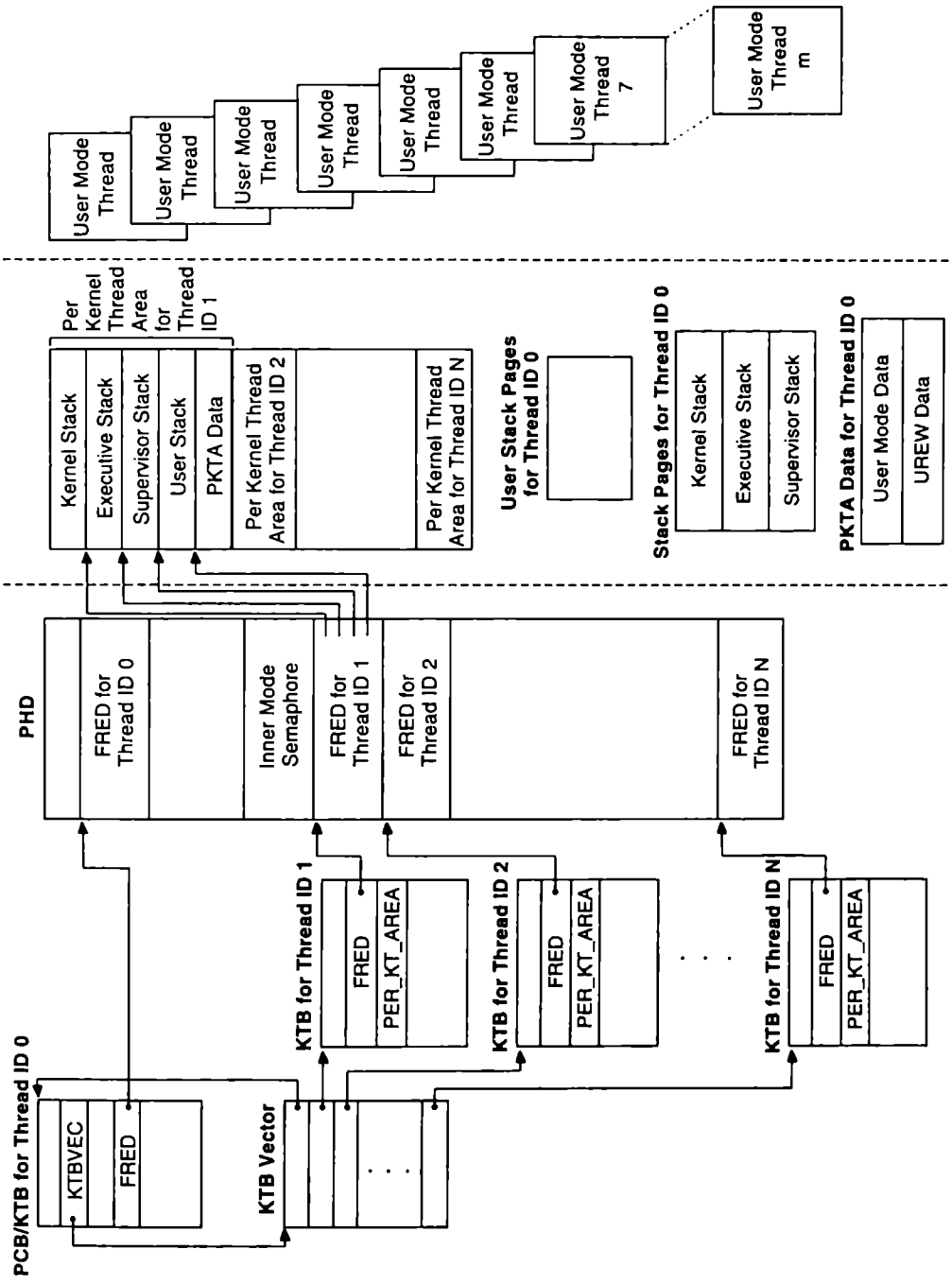



Figure 5.1
Kernel Thread Data Structures

was designed to overlay the PCB. That is, each structure is sparse, with some fields designated PCB fields and others KTB fields. Newly defined fields specific to multithread support are at the end of the KTB. Because many KTB fields retain their symbolic PCB names, the design minimized source changes to executive modules that reference fields in only the initial thread's KTB.

Figure 5.2 shows the layout of the PCB and KTB. KTB fields are shaded.

The longwords STS and STS2 are unusual in that they each represent fields in both structures. Some STS and STS2 bits are processwide and are part of PCB$L_STS and PCB$L_STS2. Others are kernel-thread-specific and are part of KTB$L_STS and KTB$L_STS2.

Each time a multithreaded process creates a new kernel thread, a KTB is allocated from nonpaged pool.

The addresses of all KTBs associated with the process are recorded in a nonpaged pool array pointed to by PCB$L_KTBVEC. The array is indexed by kernel thread ID. CTL$GL_MULTITHREAD contains the number of array elements. CTL$GL_KT_COUNT contains the number that actually point to KTBs, that is, the number of existing kernel threads. The first element in the array, the cell CTL$GL_INITIAL_KTB, and the field PCB$L_INITIAL_KTB each contain the address of the KTB for the initial kernel thread, that is, the address of the PCB.

CTL$GL_KT_HIGH and PCB$L_KT_HIGH each contain the number of the highest KTB vector entry in use, counting from the value 1. The number represents a highwater mark used when multiple entries of the KTB vector must be scanned.

Figure 5.3 shows the relations among these structures for an example process with a value of 6 in CTL$GL_MULTITHREAD and a value of 3 in CTL$GL_KT_COUNT. In this example, CTL$GL_KT_HIGH would contain the value 3.

For a single-threaded process, PCB$L_KTBVEC contains the address of the field PCB$L_INITIAL_KTB. That is, the table consists of a single longword within the PCB. CTL$GL_MULTITHREAD contains the value 0. Figure 5.4 shows the PCB/KTB for a single-threaded process.

### 5.3.3 Floating-Point Register and Execution Data Structure (FRED)

The FRED for the initial kernel thread is contained in the fixed part of the PHD. When multithreading is initiated in a process, the PHD is expanded by a page for the FREDs of other kernel threads. The page is owned by kernel mode with a protection of user read, executive write (UREW). PHD$L_FREDOFF contains the longword offset from the beginning of the PHD to this page. If the process and its PHD are outswapped, the FRED page is outswapped as part of the PHD. Figure 5.5 shows the layout of FREDs in the PHD.

**Figure 5.2**
Layout of a Process Control Block (PCB) and Kernel Thread Block (KTB)

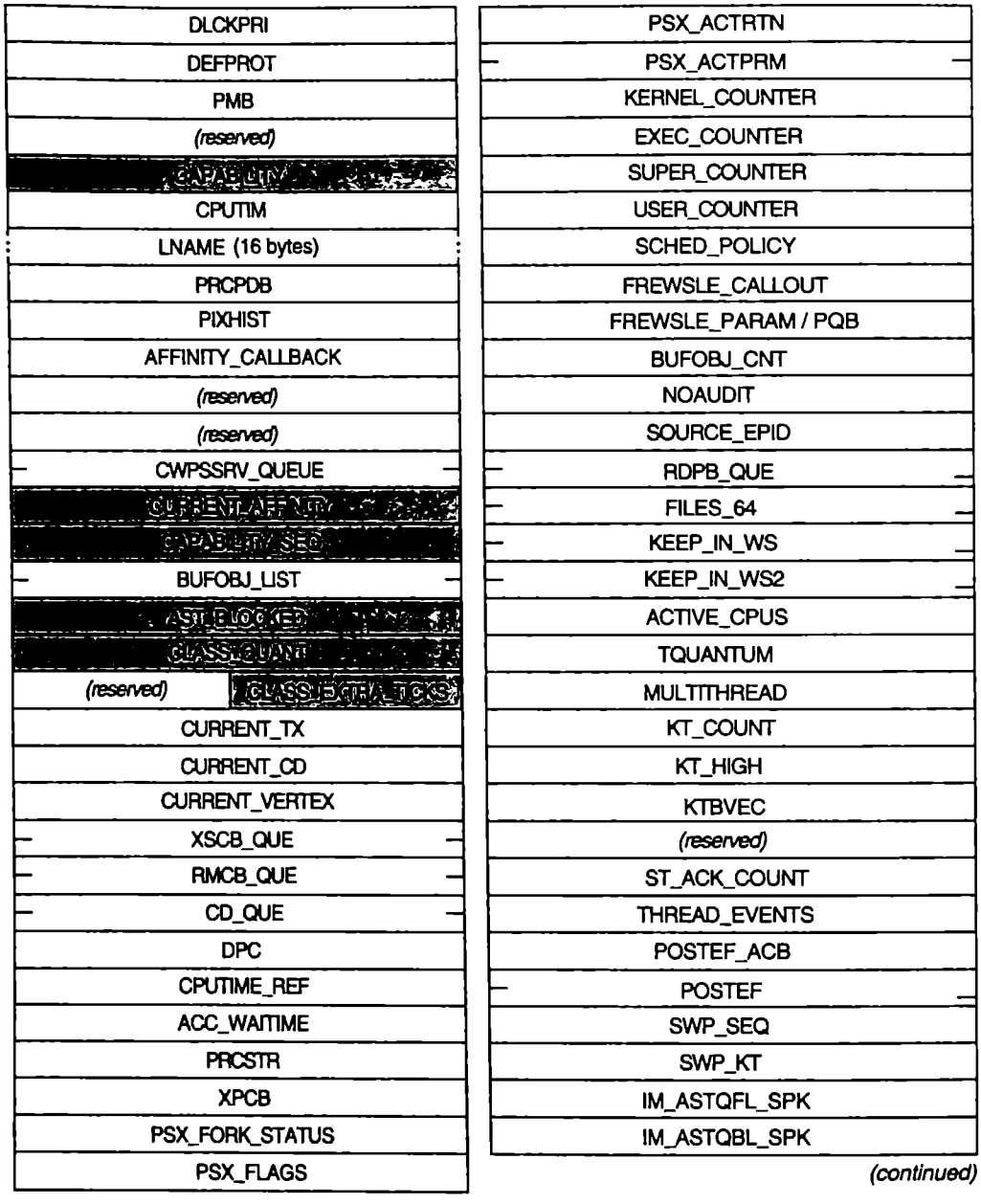
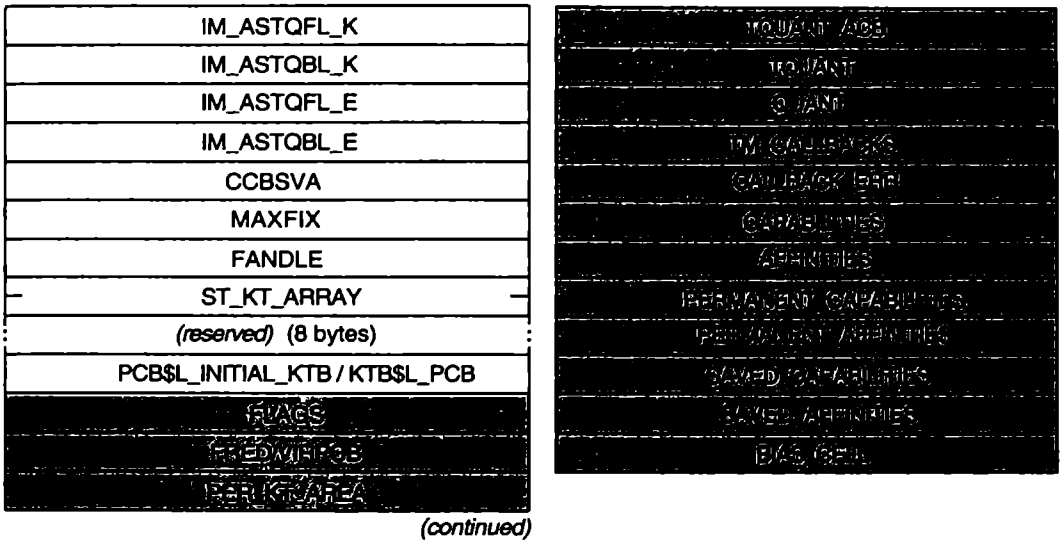



**Figure 5.2** ***(continued)***




(continued)

Figure 5.2 *(continued)*

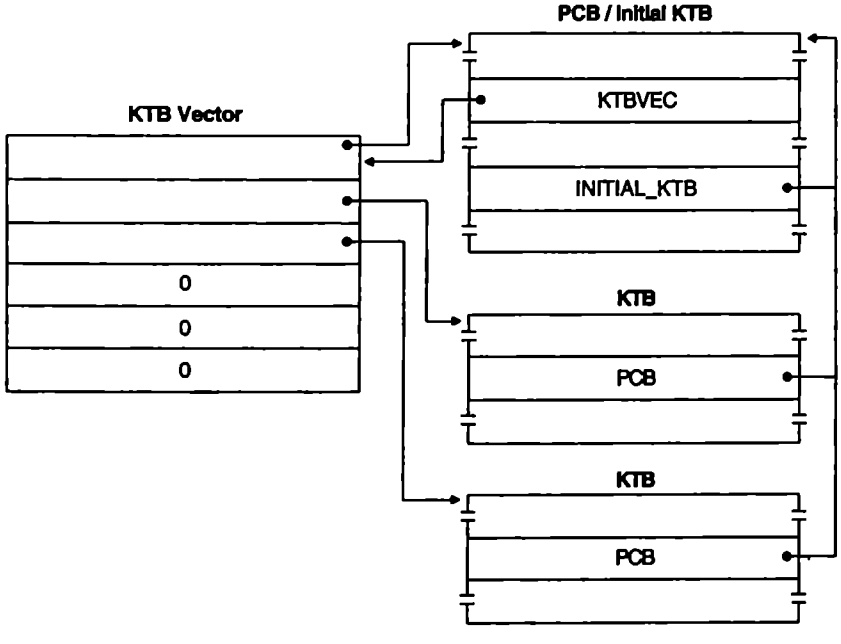



**Figure 5.3**
PCB, KTBs, and KTB Vector for a Multithreaded Process

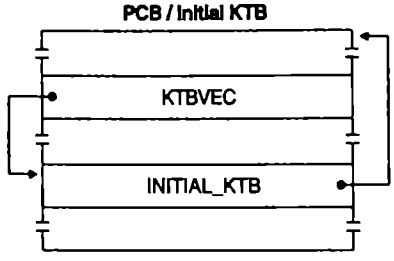



Figure 5.4
PCB, Initial KTB, and KTB Vector for a Single-Threaded Process

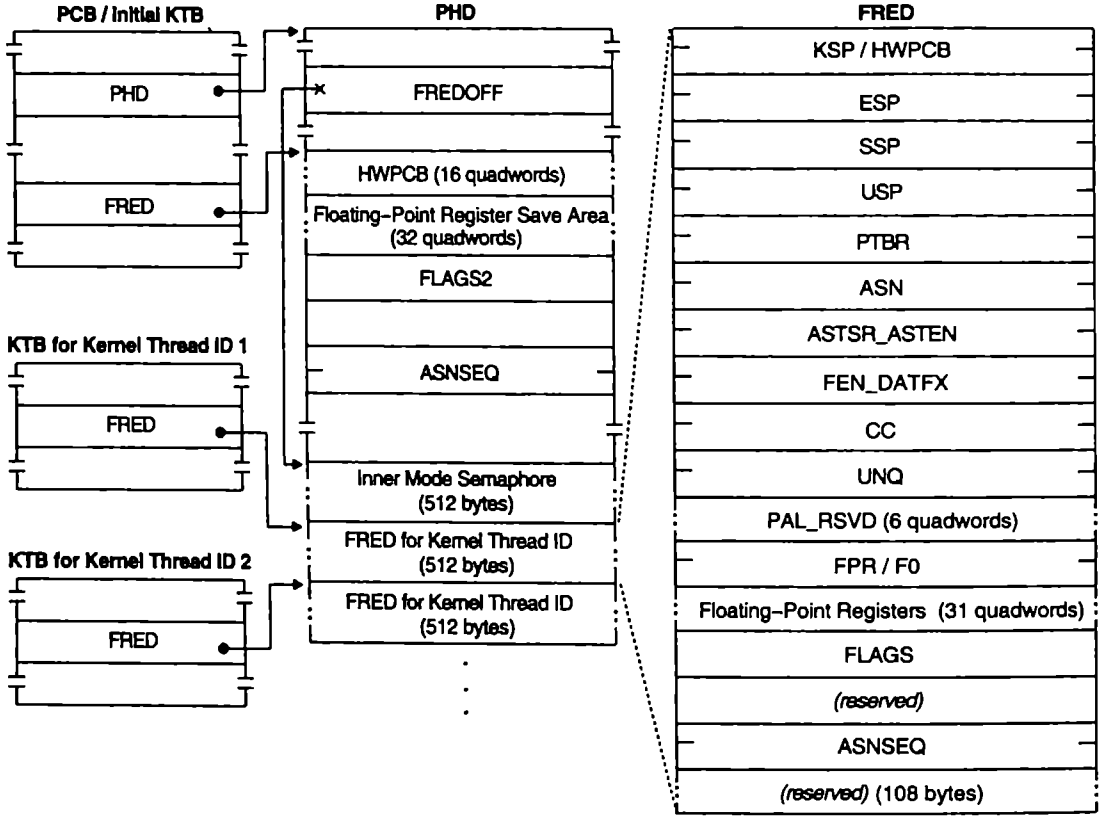



Figure 5.5
Layout of FREDs in the PHD

Like the rest of the PHD, the FRED page is double-mapped and can be accessed through both system space and P1 space addresses. Most kernel thread executive code accesses the FRED page through its P1 space addresses because they do not change. The system space addresses can only be used from code that blocks rescheduling and swapping, because if the process is outswapped, its PHD may be inswapped to a different balance set slot.

The process's inner mode semaphore is in the first 512-byte block in the FRED page. The inner mode semaphore serializes the inner mode execution of multiple kernel threads within the same process (see Section 5.8.1). An 8,192-byte page can accommodate FREDs for 15 additional kernel threads, the maximum number supported.

All the FREDs for execution contexts within a process are in adjacent PHD virtual addresses within the same physical page. This organization makes it possible for kernel mode code quickly to determine the ID of its kernel thread with several instructions rather than through a system service request. The following BLISS statements are excerpted from OpenVMS executive code:

```
pcbb = pal_mfpr_pcbb ;
my_id=(.pcbb AND ((pcb$k_max_kt_count*fred$k_length)-1)) ^-9 ;
```

These instructions get the physical address of the current hardware privileged context block (HWPCB), clear all the high-order bits except those that represent an offset into the array of FREDs, and then divide by 512, the length of a FRED.

Each FRED consists of a HWPCB followed by space in which the floating-point registers can be saved at context switch, plus several other fields. FRED$L_FLAGS replicates PHD$L_FLAGS2. In particular, it contains the AST_PENDING bit (see Chapter *ASTs*) and a software copy of the floating-point enable register. FRED$Q_ASNSEQ contains the sequence number associated with the thread's current address space number (see Chapter 2).

### 5.3.4 Per Kernel Thread Area (PKTA)

Each kernel thread has a private area in P1 space, whose address is stored in KTB$L_PER_KT_AREA. The area includes space for stacks and the PKTA data structure.

Kernel mode stack pages have protection supervisor read, kernel write (SRKW) and owner kernel mode, and are locked into the process working set list. The executive mode stack pages have protection supervisor read, executive write (SREW) and owner executive mode; the supervisor mode stack pages have protection user read, supervisor write (URSW) and owner supervisor mode; and the user mode stack pages have protection user read, user write (URUW) and owner kernel mode. The user mode stack's default size of eight pages can be overridden when multithreading is initiated for a process.

A private area for a kernel thread other than the initial one is allocated when the thread is created. The size of the area is the sum of user mode stack size, the SYSGEN parameter KSTACKPAGES, executive and supervisor mode stacks of two pages each, one data page, and one guard page. When multithreading is initiated in the process, address space is reserved for PKTAs for the maximum number of kernel threads the process can have. The cell CTL$GQ_KT_REGION_BASE contains the virtual address of the private area for thread ID 1, and CTL$GQ_KT_REGION_ID contains the ID of the region. Once created, a kernel thread private area is not deleted until multithreading is terminated in the process.

The PKTA data structure begins at the high-address end of the user stack and continues across a page boundary. This allows some fields in it to be writable from user mode while others are writable only from executive and kernel modes. The 16-element array CTL$GQ_PKTAVEC, indexed by kernel thread ID, contains the addresses of the PKTA data structures.

The initial thread's private area is permanent and defined largely by compilation of the SHELLxK module that defines the permanent part of P1 space. Its area is structured differently from those of other threads in that the stacks and data structure are not adjacent. The initial thread's PKTA begins at CTL$A_INITIAL_PKTA, and its stacks are located through the arrays CTL$AQ_STACK and CTL$AQ_STACKLIM. Furthermore, its user stack is not a fixed size; it can expand on demand until it would encroach on the address space reserved for other kernel threads' private areas.

Figure 5.6 shows the organization of this area and the layout of the PKTA data structure.

PKTA$Q_KTB contains the address of the KTB associated with this execution context; PKTA$L_PID, its PID; and PKTA$L_EPID, its EPID.

PKTA$Q_PHD contains the P1 space address of the process's PHD.

PKTA$Q_KSTACK_BASE points to the high-address end of the kernel stack for this execution context, that is, the initial value for the kernel mode stack pointer. Starting at PKTA$Q_STACK is a four-quadword array whose elements contain the initial values of the four per-process stack pointers. An array element is indexed by access mode. A similar array, starting at PKTA$Q_STACKLIM, contains the lowest limit of each stack.

The user-writable fields in the PKTA, described in the paragraphs that follow, are intended for communication between the OpenVMS executive and a user mode thread manager, in particular, the DECthreads thread manager.

PKTA$Q_TM_FLAGS contains user mode thread manager flags. The only one currently defined is PKTA$V_TM_ACTIVE, which, when set, means that the user mode thread manager has been or is about to be called in the context of this kernel thread and should be considered active.

PKTA$Q_IEEE_FP_CONTROL contains the thread-specific contents of the IEEE exceptions control register.

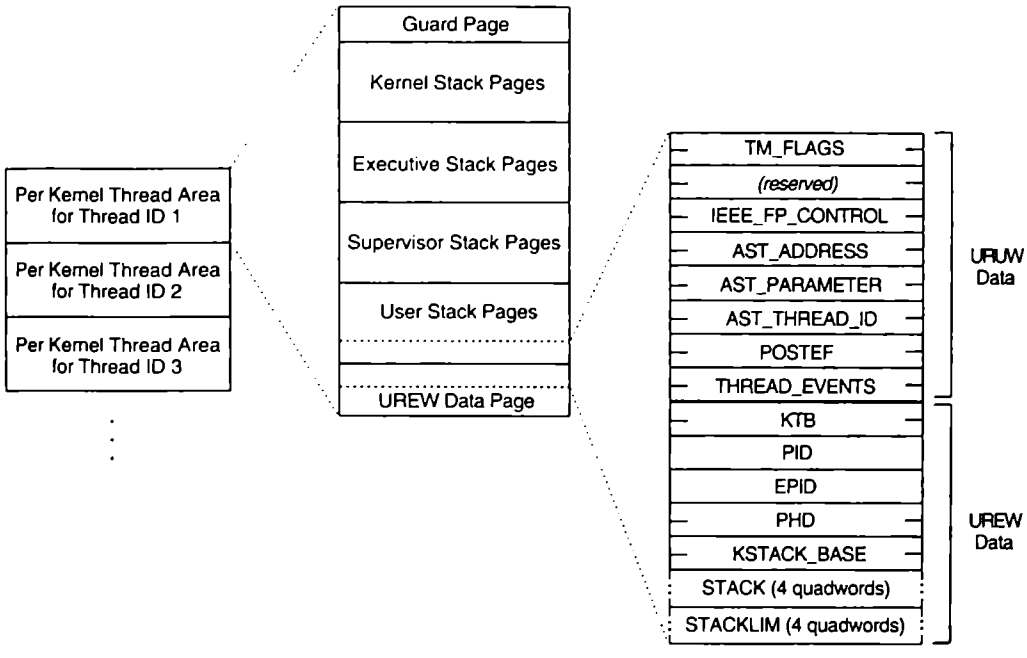



**Figure 5.6**
Layout of Per Kernel Thread Area (PKTA)

PKTA$Q_AST_ADDRESS, PKTA$Q_AST_PARAMETER, and PKTA$Q_AST_THREAD_ID communicate data between the executive and a user mode thread manager during a user mode asynchronous system trap (AST) delivery upcall (see Section 5.8.2.3).

PKTA$Q_POSTEF and PKTA$Q_THREAD_EVENTS are used to communicate data between the executive and a user mode thread manager during an event flag setting upcall (see Section 5.9.3).

## 5.4 TRANSITIONS TO AND FROM MULTITHREADING

A process is created with a single kernel thread and with multithreading disabled. To initialize the data structures to represent additional kernel threads, code running in the process must request the $INIT_MULTITHREAD system service, described in Section 5.4.1. The DECthreads run-time library requests this service during its image initialization.

After initiating multithreading, a user mode thread manager requests the $REGISTER_THREAD_MANAGER system service, described in Section 5.4.2. The DECthreads run-time library requests this service during image initialization. While an image linked with the DECthreads run-time library is activated, user mode multithreading is enabled. It remains enabled until the main image exits and is run down.

When the main image in a multithreaded process is run down, the OpenVMS executive requests the $END_MULTITHREAD system service, described in Section 5.4.3. The $END_MULTITHREAD service requests the $DEREGISTER_THREAD_MANAGER system service, described in Section 5.4.4.

The system services that initiate and terminate multithreading and register and deregister a user mode thread manager are intended for use by DECthreads and other OpenVMS components and are unsupported for other use.

### 5.4.1 Initiation of Multithreading

The $INIT_MULTITHREAD system service procedure, EXE$INIT_MULTITHREAD in module TM_SUPPORT, runs in kernel mode. It takes the following steps to initiate multithreading in the process:

1. It examines CTL$GL_MULTITHREAD to see whether the process is already multithreaded and, if so, returns the error status SS$_MTHREADACTIVE.
2. It checks that the output arguments are accessible and, if not, returns the error status SS$_ACCVIO.
3. It checks that the requested user mode stack size is within 0 and 128 pages and, if not, returns the error status SS$_BADPARAM. Specifying a value of 0 results in the default value of eight pages.
4. It checks whether the SYSGEN parameter MULTITHREAD is 0 and, if so, returns the error status SS$_NOMULTITHREAD. A value larger than 0 enables multithreading systemwide.
5. It checks whether the process is a Portable Operating System Interface (POSIX) process and, if so, returns the error status SS$_UNSUPPORTED. Use of the kernel thread mechanism with POSIX is not supported.
6. EXE$INIT_MULTITHREAD establishes the maximum number of kernel threads the process can have by minimizing the requested number against the value of MULTITHREAD. A requested value of 0 defaults to the value of MULTITHREAD.
7. It raises interrupt priority level (IPL) to 2 to block AST delivery.
8. If the maximum number of kernel threads the process can have is greater than 1, it allocates nonpaged pool for a KTB vector and zeros it.
9. It tests whether the PHD page at longword offset PHD$L_FREDOFF already exists and, if not, expands the PHD to include a FRED page.
10. It clears the FRED page, initializing the inner mode semaphore and associated data.
11. To determine the size of the PKTA, EXE$INIT_MULTITHREAD sums the requested user mode stack size, the SYSGEN parameter KSTACKPAGES, executive and supervisor mode stacks of two pages each, one data page, and one guard page.

12. It requests the $CREATE_REGION_64 system service to create a region in P1 space with a PKTA area for the maximum number of kernel threads the process can have. If the service fails, EXE$INIT_MULTITHREAD deallocates the KTB vector and returns $CREATE_REGION_64's error status to its requestor.

    If the region was created successfully, CTL$GQ_KT_REGION_BASE contains the beginning address of the region, and CTL$GQ_KT_REGION_ID, the ID associated with the region.
13. EXE$INIT_MULTITHREAD stores the size of a PKTA area in CTL$GL_KT_AREA_PAGES.
14. It stores the address of the KTB vector in PCB$L_KTBVEC and initializes its first entry to the address of the initial KTB, namely, the PCB.
15. It stores the address of the process's inner mode semaphore in CTL$AR_IM_SEMAPHORE.
16. It locks into the process working set list the URUW page of the permanent PKTA.
17. It initializes both PCB$L_MULTITHREAD and CTL$GL_MULTITHREAD to the maximum number of kernel threads the process can have. In a single-threaded process, these fields contain the value 0 or 1.
18. EXE$INIT_MULTITHREAD returns to its caller the maximum number of kernel threads, the address of the initial thread's PKTA region, and the EPID of the process. As a side effect of leaving kernel mode and returning to the access mode of the system service requestor, IPL is lowered to 0.

### 5.4.2 Registering a User Mode Thread Manager

The $REGISTER_THREAD_MANAGER system service procedure, EXE$REGISTER_THREAD_MANAGER, in module TM_SUPPORT, is called with the address of an upcall table and, optionally, a value for user mode thread quantum. The table contains addresses of thread manager procedures to be called when events occur that affect the computability of user mode threads, both events that block a thread and cause it to wait, and events that unblock a thread and cause it to be computable. Section 5.9 lists the possible upcall events.

EXE$REGISTER_THREAD_MANAGER runs in kernel mode and takes the following steps:

1. It examines CTL$GL_MULTITHREAD to see whether multithreading has been initiated within the process and, if not, returns the error status SS$_NOMULTITHREAD.
2. It examines CTL$GQ_TM_CALLBACKS to see whether a user mode thread manager has already been registered. If the field contains a nonzero value, the routine returns the error status SS$_MTHREADACTIVE.

3. If CTL$GL_KT_COUNT is greater than 1, indicating that multiple kernel threads have already been created within the process, EXE$REGISTER_THREAD_MANAGER returns the error status SS$_MTHREADACTIVE.
4. It confirms that the entire upcall table and each procedure descriptor it points to are user-readable, returning the error status SS$_ACCVIO if not.
5. It checks that the upcall table is well formed, returning SS$_BADPARAM if not. For example, each blocking upcall specified must have a corresponding unblocking upcall.
6. It raises IPL to 2 to block AST delivery.
7. EXE$REGISTER_THREAD_MANAGER allocates nonpaged pool for a copy of the upcall table and, optionally (see steps 10 and 11), two AST control blocks (ACBs). It also allocates space in the P1 allocation region for a copy of the upcall table.
8. It copies the upcall table to both nonpaged pool and the P1 allocation region, clearing any pairs of entries associated with bits set in SYSGEN parameter DISABLE_UPCALLS. This SYSGEN parameter makes possible selective disabling of upcalls for troubleshooting.
9. It records the address of the nonpaged pool copy of the upcall table in the initial kernel thread's KTB$L_TM_CALLBACKS and the address of the P1 allocation region copy in CTL$GQ_TM_CALLBACKS.
10. If the upcall table includes an entry for a thread quantum-end event, if the system service argument for thread quantum value was present, and if the corresponding DISABLE_UPCALLS bit is clear, the thread manager will be notified via an AST when thread quantum end is reached. EXE$REGISTER_THREAD_MANAGER initializes the ACB that was allocated in step 7 to describe a user mode AST with a piggyback special kernel mode AST. The user mode AST procedure is EXE$QUANTUM_AST, in module TM_SUPPORT, and the special kernel mode AST routine is SCH$THREADQ_KAST, in module RSE. This ACB will be used whenever a user mode thread running in the initial kernel thread reaches thread quantum end (see Section 5.9.2).

    EXE$REGISTER_THREAD_MANAGER records the address of the ACB in KTB$L_TQUANT_ACB and the specified quantum value in the initial kernel thread's KTB$L_TQUANT and PCB$L_TQUANTUM.
11. If the upcall table includes an entry for event flag setting and the corresponding DISABLE_UPCALLS bit is clear, the thread manager will be notified via an AST when an event flag is set. EXE$REGISTER_THREAD_MANAGER initializes the ACB that was allocated in step 7 to describe a user mode AST with a piggyback special kernel mode AST. The user mode AST procedure is EXE$SETEF_AST, in module TM_SUPPORT, and the special kernel mode AST routine is EXE$POSTEF_AST, in module POSTEF.

EXE$REGISTER_THREAD_MANAGER records the address of the ACB in PCB$L_POSTEF_ACB

12. It returns the status SS$_NORMAL. As a side effect of leaving kernel mode and returning to the access mode of the system service requestor, IPL is lowered to 0.

### 5.4.3 Termination of Multithreading

The $END_MULTITHREAD system service procedure, EXE$END_MULTITHREAD in module TM_SUPPORT, runs in kernel mode and takes the following steps to return a process to a single-threaded state:

1. It examines CTL$GL_MULTITHREAD to confirm that the process is multithreaded and, if not, returns the error status SS$_NOMULTITHREAD.
2. It checks whether CTL$GL_KT_COUNT is greater than 1 and, if not, continues with step 4.
3. It then checks whether it is running in the context of the initial kernel thread.
   - If not, it queues to the initial kernel thread a kernel mode AST that will request the $END_MULTITHREAD system service. It then calls EXE$DELETE_KERNEL_THREAD (see Section 5.5.3) to delete itself.
   - If it is running in the context of the initial kernel thread, it calls EXE$DELETE_KERNEL_THREAD for each existing kernel thread other than itself. It allocates, initializes, and queues a timer queue entry (TQE) to wake itself in 0.1 second. It acquires the SCHED spinlock and changes its scheduling state to hibernating. It returns the status SS$_WAIT_CALLERS_MODE to the change mode dispatcher, which places the kernel thread into a wait.

     When the kernel thread is awakened, it will reexecute the $END_MULTITHREAD service. If CTL$GL_KT_COUNT is still greater than 1, indicating other kernel threads still exist, it will hibernate again.
4. When EXE$END_MULTITHREAD, running in the context of the initial kernel thread, finds that CTL$GL_KT_COUNT has finally reached 1, it raises IPL to 2.
5. It calls EXE$DEREGISTER_THREAD_MANAGER (see Section 5.4.4).
6. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
7. It saves the contents of CTL$GL_MULTITHREAD (the maximum number of kernel threads the process was allowed) and clears CTL$GL_MULTITHREAD to indicate that multithreading is disabled.
8. It also clears CTL$GQ_EXIT_THREAD, PCB$L_MULTITHREAD, PKTA$Q_TM_FLAGS in the permanent PKTA, and PCB status bits PCB$V_SINGLE_THREADED and PCB$V_SOFT_SINGLE_THREAD (see Section 5.8.3).

9. EXE$END_MULTITHREAD removes each queued ACB from the common inner mode ACB queues (see Section 5.8.2.1), stores the process's PID in ACB$L_PID, and calls SCH_STD$QAST to queue the ACB to the initial kernel thread. This step is necessary because AST delivery code ignores the common inner mode queues when delivering an AST to a single-threaded process.
10. It releases the SCHED spinlock, lowering IPL to 2.
11. It unlocks from the process working set list the URUW page of the permanent PKTA.
12. If the CTL$GL_MULTITHREAD contents saved in step 7 were equal to 1, EXE$END_MULTITHREAD continues with step 15.

    Otherwise, it deallocates the KTB vector, reinitializing PCB$L_KTBVEC to the address of PCB$L_INITIAL_KTB.
13. It requests the $DELETE_REGION_64 system service to delete the PKTA region. Deleting the region results in unlocking any of its pages from the process working set list and deleting the address space for the other kernel threads' stacks.

    EXE$END_MULTITHREAD clears all but the first element of CTL$GQ_PKTAVEC.
14. It clears CTL$AR_IM_SEMAPHORE, CTL$GL_KT_AREA_PAGES, CTL$GQ_KT_REGION_BASE, and CTL$GQ_KT_REGION_ID.
15. It returns the status SS$_NORMAL, possibly to the original system service requestor but more likely to the AST procedure queued to the initial kernel thread in step 3. As a side effect of leaving kernel mode and returning to the access mode of the system service requestor, IPL is lowered to 0.

### 5.4.4 Deregistering a User Mode Thread Manager

The $DEREGISTER_THREAD_MANAGER system service procedure, EXE$DEREGISTER_THREAD_MANAGER in module TM_SUPPORT, runs in kernel mode and takes the following steps:

1. It examines CTL$GL_MULTITHREAD to see whether the process is multithreaded and, if not, returns the error status SS$_NOMULTITHREAD.
2. It examines CTL$GQ_TM_CALLBACKS to see whether a user mode thread manager has already been registered. If the field contains zero, it returns the error status SS$_NOTHREADMAN.
3. If CTL$GL_KT_COUNT is greater than 1, indicating that multiple kernel threads still exist, EXE$DEREGISTER_THREAD_MANAGER returns the error status SS$_MTHREADACTIVE.
4. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.

5. EXE$DEREGISTER_THREAD_MANAGER copies and clears both KTB$L_TM_CALLBACKS and CTL$GQ_TM_CALLBACKS to indicate there is no upcall table.
6. If a quantum end ACB was allocated as part of the pool structure, it clears PCB$L_TQUANTUM, KTB$L_TQUANT, and KTB$L_TQUANT_ACB. If the low bit of KTB$L_TQUANT_ACB was set, indicating the ACB was in use, it removes it from the ACB queue in which it was inserted.
7. If an event flag setting ACB was allocated, it clears PCB$L_POSTEF_ACB.

   If the low bit of PCB$L_POSTEF_ACB was set, indicating the ACB was in use, it removes it from the ACB queue in which it was inserted.
8. It releases the SCHED spinlock, lowering IPL to 2, and deallocates the nonpaged pool, minimally the upcall table and also possibly two ACBs. It deallocates the P1 allocation region copy of the upcall table.
9. It returns to its caller, passing it the status from the pool deallocation in step 8. As a side effect of leaving kernel mode and returning to the access mode of the system service requestor, IPL is lowered to 0.

## 5.5 KERNEL THREAD CREATION AND DELETION

A process's initial kernel thread is created when the process is created and persists throughout the life of the process. All other kernel threads are transient: they are created in response to requests from an activated image and deleted in response to explicit requests. Any kernel threads other than the initial one are deleted when the main image exits.

Sections 5.5.1 and 5.5.3 describe the steps involved in creating and deleting a kernel thread. These system services are intended for use by DECthreads and other OpenVMS components and unsupported for other use.

Section 5.6 discusses the effects of multithreading on OpenVMS scheduling and swapping. Section 5.7 describes the implications of multiple threads' sharing the process address space.

### 5.5.1 Creating a Kernel Thread

The DECthreads thread manager requests the $CREATE_KERNEL_THREAD system service when a process has fewer than the maximum possible number of kernel threads and would benefit from having an additional kernel thread to execute user mode threads.

The $CREATE_KERNEL_THREAD system service procedure, EXE$CREATE_KERNEL_THREAD, in module TM_SUPPORT, runs in kernel mode. Its input arguments specify the first procedure to be executed in the new thread, an argument to that procedure, and the base priority for the thread. Its

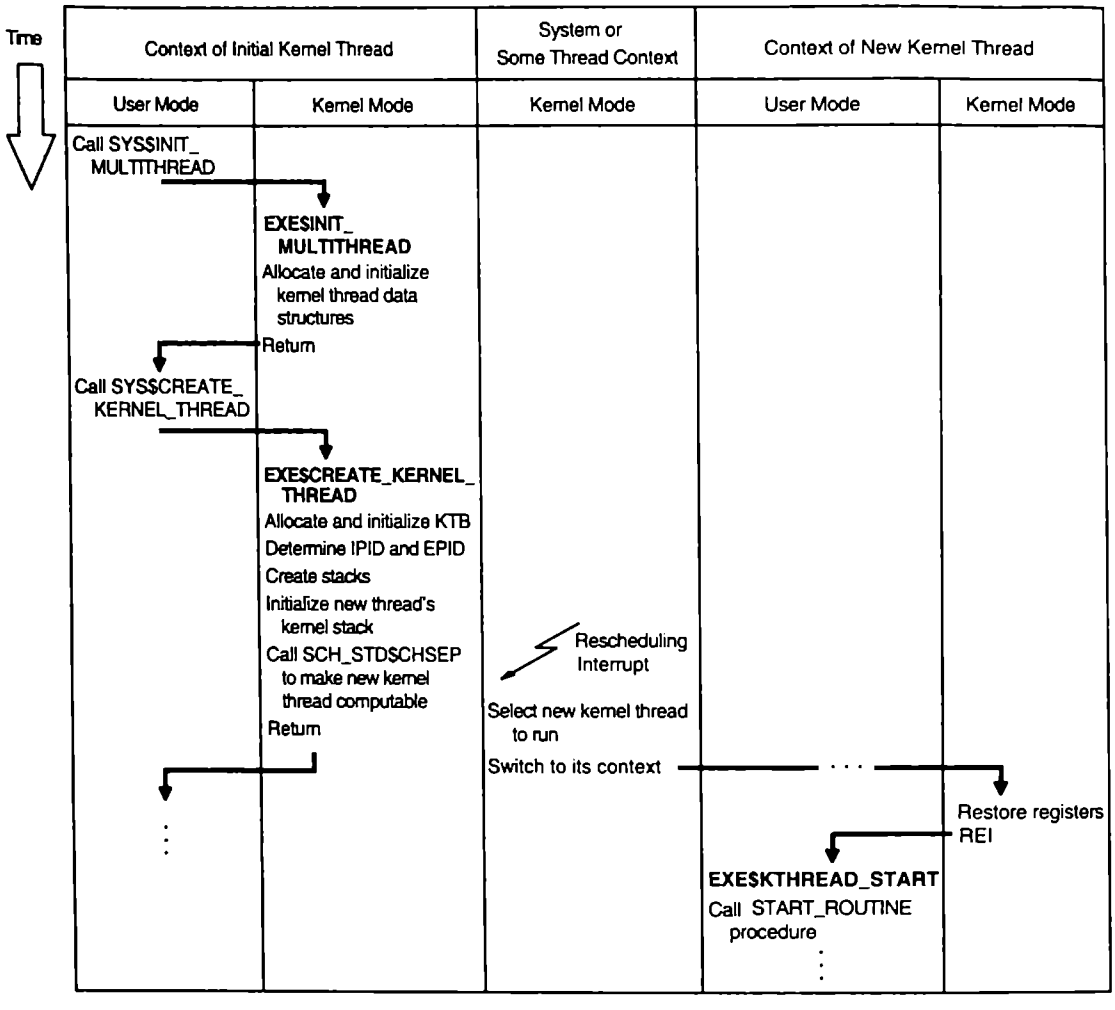



**Figure 5.7**
Main Steps in Initiating Multithreading and Creating a New Kernel Thread

output arguments specify addresses to return the EPID of the created thread and the address of its PKTA.

Figure 5.7 illustrates the sequence of events that result when an initial kernel thread initiates multithreading and creates a new kernel thread.

EXE$CREATE_KERNEL_THREAD takes the following steps to create a new kernel thread:

1. It examines CTL$GL_MULTITHREAD to confirm that the process is multithreaded and, if not, returns the error status SS$_NOMULTITHREAD.
2. It checks whether the process is already at its maximum number of kernel threads and, if so, returns the error status SS$_MAXKTHREADS.

3. It checks that the input and output arguments are accessible and, if not, returns the error status SS$_ACCVIO.
4. It checks that the base priority specified is between 0 and 63 and, if not, returns the error status SS$_BADPARAM.
5. EXE$CREATE_KERNEL_THREAD determines the ID of the current kernel thread and then the address of the current thread's KTB.
6. If a value of 0 was specified for the priority, EXE$CREATE_KERNEL_THREAD uses the current kernel thread's base priority.
7. It validates the priority against the current thread's authorized priority. If the former is higher, it checks that the process has ALTPRI privilege.
8. It raises IPL to 2 and allocates nonpaged pool for a KTB. If a user mode thread manager was registered for this process with a thread quantum, EXE$CREATE_KERNEL_THREAD also allocates pool for an ACB to deliver a quantum expiration AST to the thread manager. It clears the KTB and initializes the ACB to describe a user mode AST with a piggyback special kernel mode AST. The user mode AST procedure is EXE$QUANTUM_AST in module TM_SUPPORT, and the special kernel mode AST routine is SCH$THREADQ_KAST in module RSE. This ACB will be used whenever a user mode thread running in this kernel thread reaches thread quantum end (see Section 5.9.2).
9. EXE$CREATE_KERNEL_THREAD finds the first empty slot in the process's KTB vector and assigns it to the new kernel thread, giving the new thread a kernel thread ID corresponding to the slot index.

   The new thread's IPID is the result of adding the new thread's ID to the sequence number in the high word of the contents of CTL$GL_INITIAL_PID. EXE$CREATE_KERNEL_THREAD calls EXE_STD$CVT_IPID_TO_EPID, in module SYSPCNTRL, to determine the new thread's EPID.
10. It calculates the address of the PKTA corresponding to this kernel thread ID. It determines whether the PKTA virtual address space (see Figure 5.6) has already been created and, if not, creates it:

    a. It calls MMG$CRETVA_K, in module SYSCREDEL, to create the kernel stack pages. EXE$CREATE_KERNEL_THREAD then locks them into the process working set list. If either request fails, EXE$CREATE_KERNEL_THREAD deallocates the KTB and thread quantum ACB and returns an error status.
    b. It calls MMG$CRETVA_K to create the executive, supervisor, and user mode stack pages and the UREW PKTA data page. If any request fails, EXE$CREATE_KERNEL_THREAD deallocates the KTB and thread quantum ACB and returns the error status from MMG$CRETVA_K.
    c. It locks the UREW data page and the high-address end of the user stack into the process working set list. The URUW data cells

occupy the same page as the high-address end of the user stack. If the lock request fails, the routine deallocates the KTB and thread quantum ACB and returns the error status from the $LKWSET system service.

11. It tests whether the FRED page already exists and, if not, expands the PHD to include it.
12. It clears the new kernel thread's FRED.
13. It initializes the HWPCB in the new kernel thread's FRED, establishing each of the three inner mode stack pointers at the high-address end of its stack pages. The initial user mode stack pointer is defined so as to leave space for the URUW PKTA fields. It copies the page table base register from the initial kernel thread's HWPCB. It initializes FRED$Q_ASTSR_ASTEN to enable AST delivery to all modes.
14. EXE$CREATE_KERNEL_THREAD initializes the PKTA for the new kernel thread with the address of the new kernel thread's KTB, its IPID and EPID, and the P1 space address of the process's PHD.
15. If the IEEE$V_INHERIT bit is set in the current thread's PKTA$Q_IEEE_FP_CONTROL, the new thread's PKTA$Q_IEEE_FP_CONTROL is initialized from that of the current thread. Otherwise, it is cleared.
16. It initializes the stack limit arrays (PKTA$Q_STACK and PKTA$Q_STACKLIM) with the boundaries of the access mode stacks previously created.
17. It creates a canonical kernel stack frame (see Chapter 2) on the new thread's kernel stack, with most saved registers zeroed. The saved processor status (PS) specifies user mode, and the saved program counter (PC) and R27 specify the procedure EXE$KTHREAD_START, in module TM_SUPPORT. The saved argument registers pass the $CREATE_KERNEL_THREAD arguments START_ROUTINE and PARAMETER as well as the PKTA address and EPID of the new thread.
18. EXE$CREATE_KERNEL_THREAD initializes the KTB, for example, creating empty AST queue listheads and connecting the KTB, PKTA, PCB, and JIB. It copies capability and affinity information, user mode thread manager callback table address, authorized priority, owner, status longwords, and so on from the current thread's KTB. It initializes user mode thread quantum from PCB$L_TQUANTUM and thread quantum from the SYSGEN parameter QUANTUM.
19. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
20. It stores the system space address of the PHD and FRED in KTB$L_PHD and KTB$L_FRED and the physical address of the FRED's HWPCB in KTB$Q_PHYPCB.
21. It stores the addresses of the PKTA and KTB in the PKTA and KTB vector elements corresponding to the new thread's ID.

22. If the ID of the new thread is higher than any existing, it stores a value 1 larger than the ID in PCB$L_KT_HIGH, recording the new value also in CTL$GL_KT_HIGH.
23. It increments PCB$L_KT_COUNT, recording the new value also in CTL$GL_KT_COUNT.
24. If necessary, it updates the next sequence number in the sequence number vector (see Chapter 7) to reflect the use of this thread ID and sequence number. (This step happens only on the first use of a thread ID.)
25. It calls SCH_STD$CHSEP to make the new thread computable.
26. It releases the SCHED spinlock, lowering IPL to 0.

    Once the SCHED spinlock is released, rescheduling is possible, and the new kernel thread could begin to execute. Section 5.5.2 describes what happens when it is first placed into execution.
27. EXE$CREATE_KERNEL_THREAD returns to its requestor with the EPID of the new thread and the address of its PKTA.

### 5.5.2 Initiating Execution in a New Kernel Thread

When a new kernel thread is scheduled and placed into execution, it runs in user mode, executing code at EXE$KTHREAD_START, in module TM_SUPPORT. The procedure frame built by entering this procedure remains on the user mode stack throughout the life of the thread.

If a user mode thread manager is registered in the process (see Section 5.4.2), then EXE$KTHREAD_START sets PKTA$V_TM_ACTIVE in the thread's PKTA$Q_TM_FLAGS.

It calls the procedure specified by the $CREATE_KERNEL_THREAD system service argument START_ROUTINE. Typically, that procedure is part of the DECthreads run-time library and initializes the DECthreads "virtual processor" data structure associated with the kernel thread. Section 5.10.1 provides further details.

If the procedure specified by the START_ROUTINE argument ever returns, EXE$KTHREAD_START requests the $DELETE_KERNEL_THREAD system service to delete this thread.

In the unlikely event that $DELETE_KERNEL_THREAD returns as the result of some error, EXE$KTHREAD_START generates a fatal bugcheck. A fatal user mode bugcheck should result in image exit.

### 5.5.3 Deleting a Kernel Thread

The $DELETE_KERNEL_THREAD system service is requested to delete a kernel thread. The DECthreads run-time library does not itself request this service and instead idles unused kernel threads (see Section 5.10). The service is requested by the OpenVMS executive.

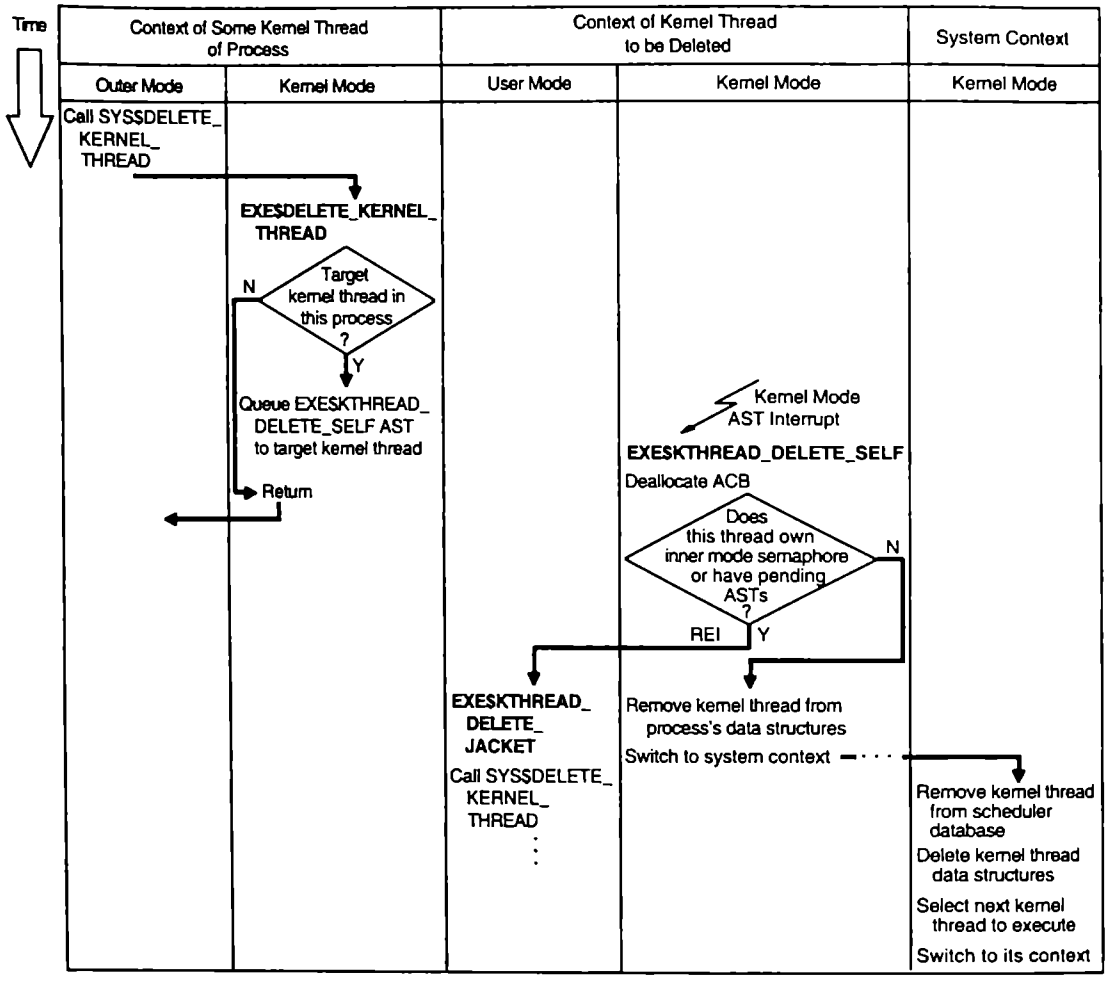



**Figure 5.8**
Main Steps in Deleting a Kernel Thread

When multithreading is being terminated in a process, the $END_MULTITHREAD system service requests $DELETE_KERNEL_THREAD to delete each kernel thread in the process except the initial kernel thread.

If the user-specified procedure indicated by the START_ROUTINE argument returns to its caller, EXE$KTHREAD_START, the latter requests the $DELETE_KERNEL_THREAD system service.

A kernel thread can delete itself or any other thread within the same process except the initial kernel thread. Thread deletion occurs in three stages: first, in the context of the kernel thread requesting the deletion; second, in the context of the thread being deleted; and finally, in system context. Figure 5.8 shows this sequence.

5.5.3.1 **Thread Deletion in Context of Caller.** The $DELETE_KERNEL_THREAD system service procedure, EXE$DELETE_KERNEL_THREAD in module

TM_SUPPORT, runs in kernel mode. It is called with one optional argument, the EPID of the thread to be deleted. If the argument is omitted, the current kernel thread is deleted.

The service procedure takes the following steps:

1. It examines CTL$GL_MULTITHREAD to see whether the process is multithreaded and, if not, returns the error status SS$_NOMULTITHREAD.
2. If the EPID was specified, EXE$DELETE_KERNEL_THREAD confirms that it identifies a kernel thread that is part of the current process. If not, the procedure returns the error status SS$_NODELETE.
3. EXE$DELETE_KERNEL_THREAD gets the address of the target thread's KTB and its PID. If the target kernel thread is the initial kernel thread, the procedure returns the error status SS$_NODELETE.
4. If the specified kernel thread is already in a state of delete pending (KTB$V_DELETE_PENDING is set in KTB$L_FLAGS), the procedure returns SS$_NORMAL.
5. Otherwise, it raises IPL to 2 and allocates nonpaged pool for an ACB. It initializes the ACB to describe a thread-safe (see Section 5.8.1) kernel mode AST with a piggyback special kernel mode AST routine, EXE$KTHREAD_DELETE_SELF, and AST procedure IOC$RETURN. It queues the ACB to the target kernel thread.

   Section 5.5.3.2 describes kernel thread deletion in the context of the thread being deleted.
6. EXE$DELETE_KERNEL_THREAD acquires the SCHED spinlock, raising IPL to IPL$_SCHED, and sets KTB$V_DELETE_PENDING in the target thread's KTB. It queues the ACB to the target thread.
7. It checks whether the process is in the single-thread state or in a state of hard suspension. If either is true, EXE$DELETE_KERNEL_THREAD reports a resume scheduling event for the target kernel thread to ensure that it will be in a computable state for the AST.
8. It releases the SCHED spinlock, lowering IPL to 2, and returns SS$_NORMAL. IPL is kept at 2 to block AST delivery until after the system service dispatcher has released the inner mode semaphore. This is an attempt to prevent other executive and kernel mode ASTs from being queued to a thread that is deleting itself.

5.5.3.2 **Thread Deletion in Context of Kernel Thread Being Deleted.** Thread deletion continues in a special kernel mode AST routine that executes within the context of the thread to be deleted.

EXE$KTHREAD_DELETE_SELF, in module TM_SUPPORT, takes the following steps:

1. It deallocates the ACB.

2. It checks whether it is running in the initial kernel thread and, if so, returns SS$_NORMAL to the AST delivery code; the initial thread cannot be deleted except when the entire process is deleted.
3. It gets its own KTB address and tests whether KTB$V_DELETE_PENDING is already set. In the unlikely circumstance that the bit is already set, the routine returns SS$_NORMAL.
4. It removes any pending wake requests for this kernel thread from the timer queue.
5. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
6. EXE$KTHREAD_DELETE_SELF checks whether this kernel thread had initiated a single-thread request (see Section 5.8.3) and, if so, cancels the request and resumes any stalled kernel threads.
7. It checks whether the process's inner mode semaphore is owned by the current thread. If so, executive and kernel mode ASTs may have been queued to this kernel thread. Before it releases the semaphore, it must REI to an outer mode so that the ASTs can be delivered (see Section 5.8.1). To do so, it takes the following steps:

   a. It sets the owner depth in the inner mode semaphore to zero but leaves the owner field intact.
   b. It enables AST delivery to all modes, sets the AST summary register to indicate pending ASTs in all modes, and clears KTB$L_ASTACT to indicate no active ASTs in any mode, thereby effecting a Clear AST ($CLRAST) system service request.
   c. It clears KTB$V_DELETE_PENDING.
   d. If the IMSEM$V_CHECK_UPCALL bit is set, it clears the bit to prevent an upcall to the user mode thread manager when the semaphore is released.
   e. It checks whether IMSEM$V_TM_WAITRS is set in the inner mode semaphore and, if so, clears it.
   f. It releases the SCHED spinlock, lowering IPL to 0.
   g. It resets the user mode stack pointer for this kernel thread to its base and calls EXE$REI_INIT_STACK to return to user mode to execute EXE$KTHREAD_DELETE_JACKET.

      Before executing any user mode code, the kernel thread is interrupted to deliver the pending ASTs. After they execute, the AST interrupt is dismissed and control returns to user mode.
   h. In user mode, EXE$KTHREAD_DELETE_JACKET requests the $DELETE_KERNEL_THREAD system service to delete itself. The service reexecutes from step 1 in Section 5.5.3.1. After having queued a kernel mode AST to itself, the service returns to the system service dispatcher, which releases the inner mode semaphore. Thus, when EXE$KTHREAD_DELETE_SELF reexecutes, it finds the semaphore unowned by the current thread (at the beginning of step 7 in this section).

8. If the inner mode semaphore is not owned by the current thread, EXE$KTHREAD_DELETE_SELF checks whether any ASTs are pending for this thread (nonzero KTB$L_AST_PENDING). If there are any, it enables AST delivery to all modes, sets the AST summary register to indicate pending ASTs for all modes, and clears KTB$L_ASTACT to indicate no ASTs active in any mode. It clears KTB$V_DELETE_PENDING, releases the SCHED spinlock, and resets the user stack pointer to its base. It then calls EXE$REI_INIT_STACK to return to user mode to execute EXE$KTHREAD_DELETE_JACKET.

   Returning to user mode enables the ASTs to be delivered. When EXE$KTHREAD_DELETE_JACKET executes, it requests the $DELETE_KERNEL_THREAD system service to delete itself. The service reexecutes from step 1 in Section 5.5.3.1.
9. If the inner mode semaphore is not owned by the current thread and no ASTs are pending for this thread, EXE$KTHREAD_DELETE_SELF continues deleting itself. It clears its entry in the KTB vector and the PKTA pointer to the KTB.
10. It decrements both PCB$L_KT_COUNT and CTL$GL_KT_COUNT to indicate one less kernel thread.
11. If its ID is the highest of any existing threads, it resets the thread ID highwater mark in CTL$GL_KT_HIGH and PCB$L_KT_HIGH to 1 more than the previous thread ID recorded in those fields.
12. It calls SCH$DELETE_CALLBACK, in module SCHEDULER. When deleting a kernel thread other than the initial kernel thread, SCH$DELETE_CALLBACK clears the bit corresponding to the current CPU in PCB$L_ACTIVE_CPUS to indicate the process has no kernel thread currently executing on that CPU and decrements PCB$Q_ONCPUCNT to indicate one less active kernel thread. If there are no other active kernel threads in the process, SCH$DELETE_CALLBACK stores the CPU ID in PCB$L_PRVCPU and copies the current address space number and associated sequence number from the FRED to PCB$Q_PRVASN and PCB$Q_PRVASNSEQ (see Chapter 2). It swaps to system context and continues execution, in EXE$KTHREAD_DELETE_CALLBACK.

5.5.3.3 **Thread Deletion in System Context.** Running in system context with the SCHED spinlock held, EXE$KTHREAD_DELETE_CALLBACK takes the following steps:

1. It gets the address of the KTB of the kernel thread being deleted from the per-CPU database.
2. It calls SCH$CLEAR_AFFINITY_BINDINGS, in module CAPABILITY, to release the deleted thread's hard affinity for a CPU.
3. It checks whether the thread was created with a timer quantum ACB and, if so, whether the ACB is currently queued to the KTB. If it

is, EXE$KTHREAD_DELETE_CALLBACK clears the in-use bit and removes it from the queue.

4. It scans the AST queues in the KTB for any ACBs still enqueued. It removes each it finds, replaces its ACB$L_PID contents with the PID of the initial thread, and queues the ACB to the process's initial kernel thread.
5. It deallocates the KTB and, as appropriate, the timer quantum ACB.
6. It returns to SCH$DELETE_CALLBACK, which clears the bit representing the CPU's ID in the longword corresponding to the priority of the kernel thread in the array at SCH$AL_CPU_PRIORITY. If there are no other kernel threads at this priority current on any CPU, it also clears the bit corresponding to that priority in SCH$GQ_ACTIVE_PRIORITY. Chapter 10 provides more information on SCH$DELETE_CALLBACK's subsequent actions.

## 5.6 SCHEDULING

Kernel threads in the same process are fairly independent of one another, except that they share the process's address space and quotas and limits, which are still maintained on a processwide basis.

Security profile features, including privileges, user identification code (UIC), and rights, currently apply to the process as a whole. Event flags and I/O requests are also handled on a processwide basis.

The OpenVMS scheduling model for processes with multiple kernel threads is largely the same as it is for single-threaded processes. The sections that follow discuss the aspects of scheduling that are specific to multithreading support.

### 5.6.1 Priority and Scheduling of Kernel Threads

The scheduling priority of each kernel thread rises and falls in response to thread execution, waits, and events that satisfy wait conditions. All priority fields formerly in the PCB are now in the KTB. The priority of each kernel thread in a process is independent of the others' except insofar as, by default, they share the same base priority. This change is consistent with the general OpenVMS scheduling model and also allows for more flexibility for a user mode thread manager. For example, a user mode thread manager might define different base priorities for different kernel threads and schedule user mode threads on specific kernel threads.

A DECthreads application can specify priority ranges and a scheduling policy for each user mode thread. Consult the *Guide to DECthreads* for further details.

In general, OpenVMS selects the highest priority computable resident scheduling entity for execution on an available processor, as it has tradi-

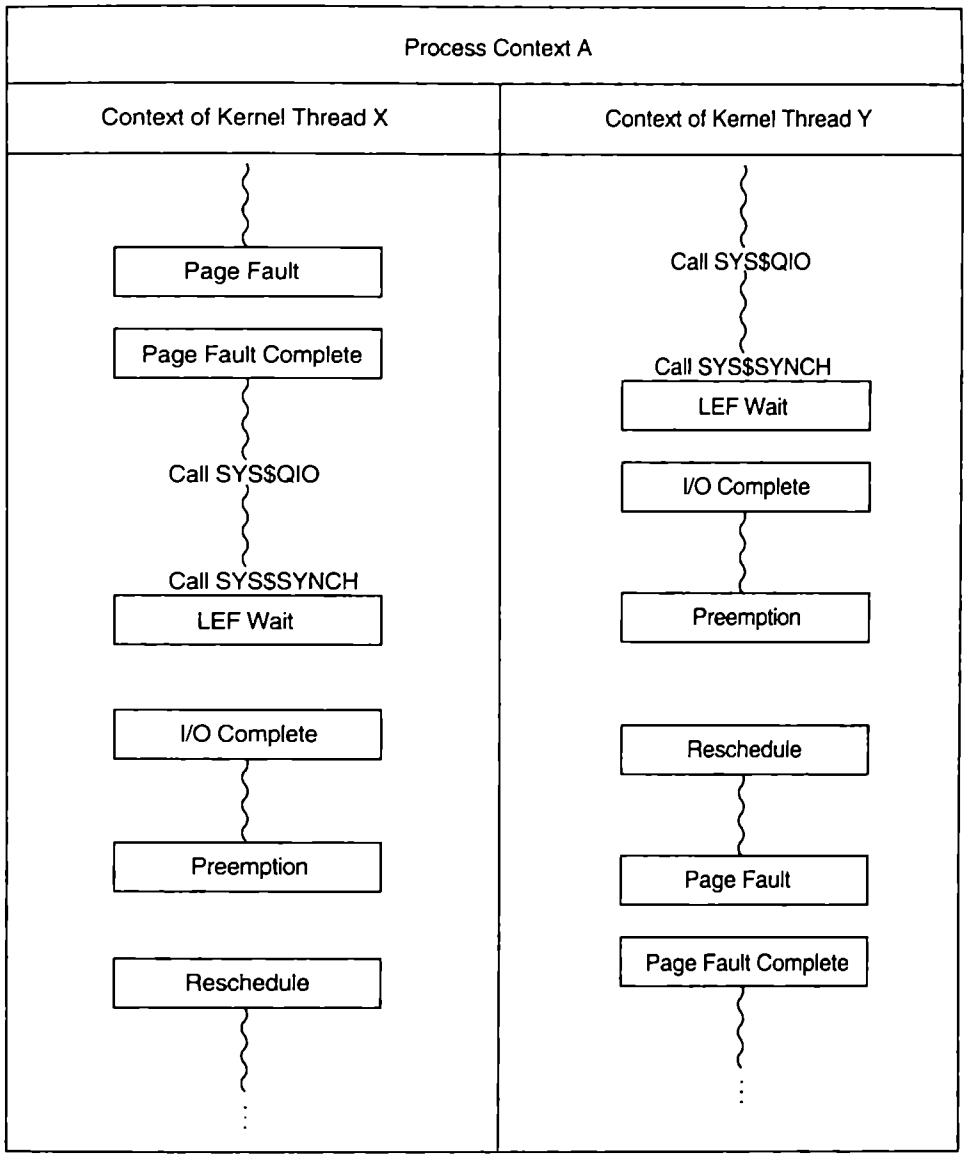



**Figure 5.9**
Example Kernel Thread Flows

tionally. However, in OpenVMS Alpha Version 7.0, the scheduling entity is the kernel thread rather than the process.

Figure 5.9 illustrates the independence of kernel threads. It shows an example process with two kernel threads running concurrently on two nodes of an SMP system. The wavy lines represent the execution of arbitrary application code within the kernel thread. Scheduling events are represented by rectangles. The two threads' execution, waits, and rescheduling are largely independent. Section 5.8 discusses the conditions under which one kernel thread's execution can block that of another, namely when one kernel thread

tries to execute inner mode code while another kernel thread in the same process is already in an inner mode.

Kernel thread X executes instructions until it incurs a page fault and is placed by OpenVMS scheduling code into a page fault wait. When the faulting page is made valid, scheduling code makes the kernel thread computable and later selects the kernel thread for execution. The kernel thread continues, requesting an I/O operation and then the Synchronize ($SYNCH) system service to wait for the I/O to complete. It is placed into a local event flag wait until the I/O completes, at which time scheduling code makes the kernel thread computable again and selects it to execute. It continues to execute but is preempted by some higher priority kernel thread. Later, it is selected to execute again.

Kernel thread Y executes, requests an I/O operation and then the $SYNCH system service to wait for the I/O to complete, and is placed into a local event flag wait. When the I/O completes, the kernel thread is made computable and selected for execution. It continues executing but is preempted. Later, it is rescheduled and executes until it incurs a page fault and is placed into a wait. After the page is made valid, the kernel thread's execution continues.

If an application running in a process has registered a user mode thread manager and requested upcalls, the OpenVMS executive makes an upcall to the thread manager whenever a kernel thread in that process would be placed into one of the following wait states in user mode: local event flag wait, hibernating, page fault wait, or collided page wait. The user mode thread manager is responsible for switching a runnable user mode thread into the kernel thread's context to prevent the kernel thread's being waited. The thread manager is also responsible for unblocking stalled user mode threads in response to executive upcalls signaling unblocking events. Section 5.9.1 has further information.

There is no user mode thread manager in this example process. Section 5.10.5 discusses a contrasting example, in which the DECthreads thread manager schedules multiple user mode threads to run on a kernel thread.

The Suspend Process ($SUSPND) system service affects the entire process; each kernel thread is placed into a state of suspension. All other wait states affect only a single kernel thread, whether it is the only kernel thread in a single-threaded process or one of many in a multithreaded process.

### 5.6.2 Kernel Thread Quantum

When a single-threaded process is inswapped or created, PCB$V_INQUAN is set in PCB$L_STS to indicate that the process is in its first quantum of CPU time and memory residence. Its initial kernel thread is initialized with a full quantum (KTB$L_QUANT gets the value of the SYSGEN parameter

QUANTUM). Such a process is not a good candidate for outswapping or swapper trimming.

When a multithreaded process is inswapped, each of its kernel threads receives a full quantum. Deductions are made from each kernel thread's quantum as the kernel thread uses CPU time and undergoes voluntary waits (see Chapter 2). PCB$V_INQUAN is cleared the first time a kernel thread in the process reaches quantum end.

Whenever a kernel thread reaches quantum end, the executive checks whether the process has exceeded a CPU time limit and, if so, queues an AST to the kernel thread to exit the current image and delete the process.

At quantum end, the executive also checks whether automatic working set list adjustment is appropriate. The checks made to determine whether the working set of a multithreaded process should be reduced are the same as those made for a single-threaded process with the additional constraint that no other kernel thread in the process may be current (PCB$Q_ONCPUCNT must be equal to 1).

### 5.6.3 Swapping and Swapper Trimming of Multithreaded Processes

One significant effect of multithreading is that the scheduling states of every kernel thread in a process are taken into account in the selection of a working set to trim or outswap. Chapter *The Swapper* describes in detail the OSWPSCHED table, which drives the selection of working sets to be trimmed and outswapped.

The swapper potentially makes several passes through the table, scanning many of the scheduler database queues on each pass, driven by the table's entries. Each kernel thread it finds in a particular scheduling state may have to meet certain criteria for its working set to be a candidate.

In the case of a process with multiple kernel threads, each of its kernel threads must be a candidate. Each must meet the criteria specified in the table entry for that scheduling state for the kind of pass the swapper is doing.

Consequently, a process with any kernel thread current on a CPU cannot be outswapped or trimmed because the current kernel thread would not be a candidate.

A single-threaded process is inswapped only if its initial kernel thread is in a computable outswapped (COMO) scheduling state and its priority is high relative to that of outswap and trim candidates. The swapper calls a scheduling routine to change the state of the initial kernel thread to computable (COM).

In contrast, a multithreaded process becomes an inswap candidate on the basis of its highest priority COMO kernel thread. The states of its other kernel threads are irrelevant except that each must potentially be changed to a resident form. For example, a hibernating outswapped (HIBO) kernel thread must be changed to hibernating (HIB).

## 5.7 IMPLICATIONS OF SHARED MEMORY

In general, a kernel thread executes independently of other kernel threads in the same process. That all threads share the same virtual address space, however, has several implications.

Even though multiple kernel threads share the same address space, they cannot all use the same translation buffer address space number (ASN) in an SMP system because ASNs are CPU-specific. Chapter 2 contains more details.

When code running in a multithreaded process on an SMP system changes a valid process-private page table entry (PTE), it must invalidate that entry in the processor's translation buffer. Moreover, it must inform all other SMP members on which other kernel threads' ASNs are active to flush the cached contents of that PTE, now stale, from their translation buffers. The field PCB$L_ACTIVE_CPUS contains a 32-bit mask with a bit set for each CPU on which a kernel thread in that process is active. An interprocessor interrupt is sent to each CPU represented in the mask. Chapters *Working Set List Dynamics* and *Symmetric Multiprocessing* contain more information.

Because multiple processors can execute multiple kernel threads of the same process and make concurrent access to process data, additional synchronization is required, both in OpenVMS code and in application code. Section 5.8 discusses OpenVMS executive synchronization techniques.

Code running in one kernel thread can request memory management system services that delete process-private address space or alter its protection. OpenVMS executive code must prevent such a change from executing concurrently with inner mode code in another kernel thread that might access the altered address space. This issue is discussed briefly in Section 5.8.

Multiple kernel threads concurrently accessing pageable process-private address space affect the movement of pages into and out of the process working set list. The side effects of this and the mechanism by which executive code can temporarily force a page into a process working set list are described in the following paragraphs.

The makeup of the dynamic part of a process's working set changes to reflect execution history: pages are faulted in as a result of code execution, and when the working set list is full, older pages are removed to make way for pages being faulted in. In previous OpenVMS versions, executive code used a combination of the following techniques to keep the working set list from changing temporarily except for explicitly faulted-in pages:

- Raising IPL to 2 to block AST delivery and execution of any other thread of execution within the process and thus prevent unforeseen replacement paging
- Setting the bit PHD$V_NO_WS_CHNG in PHD$L_FLAGS to block swapper trimming, automatic working set list adjustment, and proactive mem-

ory reclamation, all of which could cause pages to be removed from the working set list

These techniques could not work in a multithreaded process. For example, raising IPL to 2 blocks AST delivery only to that kernel thread and does not inhibit execution of other kernel threads. Consequently, these techniques have been replaced in OpenVMS Alpha Version 7.0 with two PCB fields that delimit the virtual address range whose pages should not be removed from the working set list: PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2. Each field is initialized to the value −1.

Executive code that must block removal of particular pages from the working set list acquires the MMG spinlock to synchronize access to these fields from other kernel threads (in particular, the page fault handler running in another kernel thread) and stores the starting and ending addresses in the two fields.

Chapter *Working Set List Dynamics* contains further details.

## 5.8 SYNCHRONIZATION WITHIN A MULTITHREADED PROCESS

In versions of OpenVMS Alpha prior to Version 7.0, the maximum number of active threads of execution within a process was limited by a combination of the following:

- One hardware context per process
- OpenVMS constraints on AST delivery
- Architectural constraints on access mode changes

In a single-threaded process, one hardware context per process limits the number of concurrent threads of execution to a single thread. A second asynchronous thread of the same access mode can become active as the result of an AST interrupt. Until the AST completes, however, control does not return to the interrupted thread.

OpenVMS does not interrupt AST execution to deliver another AST to the same mode. If an AST procedure is executing, the main thread cannot interrupt it or execute concurrently. An AST procedure can therefore be guaranteed uninterrupted access to data shared with a single main thread. A main thread needing uninterrupted access can block delivery of an AST of the same access mode through the Set AST Enable ($SETAST) system service. A main thread running in kernel mode can block delivery of special kernel mode ASTs by raising IPL to 2.

An AST can be interrupted by delivery of an AST for a more privileged mode. This typically does not present synchronization problems because few data structures are shared across access modes; most data is specific to a set of routines that run in one access mode.

The VAX and Alpha architectures limit the ways in which access mode can change: the CPU can change to a more privileged mode as the result of

an exception or interrupt; it can return to a less privileged mode through the instruction REI.

In a single-threaded OpenVMS environment, routines can be written with the following assumptions about synchronizing access to shared process-private data:

- Code running in kernel mode at IPL 2 cannot be interrupted by delivery of normal kernel mode ASTs or special kernel mode ASTs.
- Code running in a mode other than kernel can block execution of all ASTs of the same mode through the $SETAST system service.
- Code running in a mode other than kernel can be interrupted by a more privileged mode AST.
- Code running in a mode other than kernel can change to executive or kernel mode to execute a system service.
- Code running in one mode will not be interrupted by a routine in a less privileged mode.

In summary, in a single-threaded environment, code that executes in an inner mode and that blocks ASTs can generally assume it has exclusive access to process-private data particular to that mode.

In contrast, in OpenVMS Alpha Version 7.0, in a multithreaded process multiple threads of execution can run on multiple CPUs concurrently, each potentially requesting system services and receiving ASTs. To minimize the desynchronizing impact of these capabilities on existing kernel and executive mode code, OpenVMS Alpha provides the following mechanisms:

- A method to distinguish thread-safe inner mode code. The term *thread-safe* refers to code whose execution is unaffected by the actions of other threads within the same process. In the more limited context of the OpenVMS executive, the term *thread-safe* is applied only to executive and kernel mode code. In other words, a thread-safe inner mode procedure requires no executive assistance to synchronize with procedures running in other threads in the same process. Section 5.8.1 describes how inner mode procedures can be declared thread-safe.

  Any routine that accesses process-private address space that could be deleted or whose protection could be altered by another concurrent kernel thread is not thread-safe.
- The inner mode semaphore, a new process-private synchronization mechanism, serializes non-thread-safe inner mode execution. Section 5.8.1 describes this mechanism and its use.
- The $SINGLE_THREAD system service, described in Section 5.8.3, enables one kernel thread to stall all the others.

### 5.8.1 Inner Mode Semaphore

The inner mode semaphore is a process-specific structure by which non-thread-safe process context execution in kernel and executive mode can be restricted to one kernel thread at a time. OpenVMS restricts execution by requiring a kernel thread to acquire the semaphore when it begins to execute in kernel or executive mode and to release it when it is about to leave the mode.

An inward access mode transition is possible only through an interrupt or exception. The only interrupts handled in process context are AST delivery interrupts. The exceptions of most relevance to synchronizing multithread execution are change mode to executive and change mode to kernel exceptions. Each OpenVMS executive routine that handles one of these interrupts or exceptions acquires the inner mode semaphore, if necessary, and releases it before exiting.

Each executive or kernel mode system service and AST procedure, by default, must acquire the semaphore. It is possible, however, for a service or AST procedure to be defined as thread-safe, able to execute concurrently with other inner mode kernel threads in the process.

Each system service, whether part of an executive image or a privileged shareable image, defines a compile-time flag that specifies whether the service is thread-safe. By default the flag is clear, meaning that the service should be executed with the inner mode semaphore held. When set, the flag indicates that the service is thread-safe and does not have to acquire the inner mode semaphore.

A system service can set the flag if both of the following are true:

- It requires no synchronization to process-private data beyond any synchronization explicitly coded within it.
- It accesses no process-private address space that could be deleted or whose protection could be altered by another concurrent kernel thread.

The flag can be overridden when the system service is loaded: if the value of the SYSGEN parameter MULTITHREAD is 0 or 1, meaning that only one kernel thread within a process can execute at a time, the flag for each system service is set, implicitly making it thread-safe.

When a non-thread-safe system service is requested, the system service dispatcher checks whether the process is multithreaded (CTL$GL_MULTITHREAD is greater than 1). If so, it acquires the inner mode semaphore before continuing and releases it at system service exit. If the semaphore is already in use, the dispatcher waits the kernel thread. Sections 5.8.1.2 and 5.8.1.3 contain further information.

OpenVMS Alpha Version 7.0 ACBs contain a flag (ACB$V_THREAD_SAFE in ACB$L_FLAGS) that, when set, indicates that a kernel or executive mode AST procedure is thread-safe. By default the flag is clear. Kernel mode code

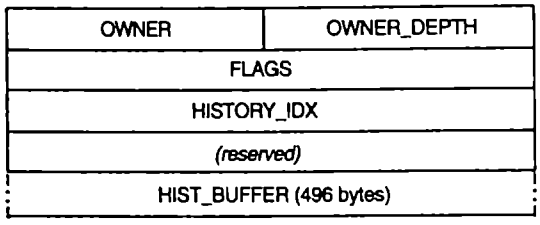



**Figure 5.10**
Layout of the Inner Mode Semaphore

that initializes an ACB to queue an AST to a kernel thread can set the flag.

When a non-thread-safe inner mode AST is being queued to a kernel thread in a multithreaded process, the executive checks the current state of the inner mode semaphore.

If the semaphore is unowned, the executive acquires it on behalf of the kernel thread. When all non-thread-safe inner mode ASTs have been delivered to the semaphore owner, AST exit code releases the semaphore.

If the semaphore is owned by another of the process's kernel threads, the AST is instead delivered to that kernel thread before it releases the semaphore.

Sections 5.8.2.2 and 5.8.2.3 have more details.

5.8.1.1 **Semaphore Data Structure.** The inner mode semaphore is created by the expansion of the PHD when multithreading is initiated in the process. CTL$AR_IM_SEMAPHORE contains the P1 space address of the semaphore. (Recall that the PHD is double-mapped into P1 space.) Figure 5.10 shows the layout of the inner mode semaphore.

IMSEM$W_OWNER identifies the kernel thread that owns the semaphore; it contains the sequence number part of that kernel thread's IPID. A value of 0 indicates no owner.

IMSEM$W_OWNER_DEPTH is the number of concurrent nested acquisitions of the semaphore. For example, if one inner mode system service calls another inner mode service, the depth would be 2. As each system service returns, the depth is decremented. A nonzero IMSEM$W_OWNER can be consistent with a depth of 0: a kernel thread retains its ownership of the semaphore after releasing all nested acquisitions if an inner mode AST has been queued to the kernel thread while it owned the semaphore. During the servicing of the subsequent AST interrupt, the semaphore will be reacquired.

IMSEM$IL_FLAGS contains several flags:

- IMSEM$V_KAST_PENDING, when set, indicates that one or more kernel mode or special kernel mode ASTs are queued to the process that require the inner mode semaphore.

- IMSEM$V_EAST_PENDING, when set, indicates one or more executive mode ASTs are queued to the process that require the inner mode semaphore.
- IMSEM$V_RS_WAITRS, when set, indicates that one or more other kernel threads of this process have been placed into resource wait for the semaphore.
- IMSEM$V_TM_WAITRS, when set, indicates that one or more user mode threads of this process have been stalled by the user mode thread manager because the semaphore was owned by another thread.
- IMSEM$V_CHECK_UPCALL, when set, indicates that the current use of the semaphore is such that when it is released, an upcall can be made to a user mode thread manager to notify it of the release. Effectively, the set bit means that when another kernel thread fails to acquire the semaphore for a system service request made from user mode, the user mode thread manager should be notified in case it can switch to another user mode thread and thus allow the kernel thread to continue execution. When the bit is clear, any kernel thread unsuccessful at acquiring the inner mode semaphore is placed into an OpenVMS executive resource wait. The bit can only be set under the following circumstances:
  - The acquisition of the semaphore is the first; that is, it is not a nested acquisition.
  - The system service was requested from user mode.
  - The requested service cannot return a wait-caller's-mode status.
  - The process has not enabled system service failure exception for the system service mode.

Most of the rest of the data structure is a ring buffer for entries identifying acquisitions and releases of the semaphore. IMSEM$IL_HISTORY_IDX contains the quadword index of the next entry to be written in the buffer. Information is recorded in the ring buffer only when the system is running the monitoring version of the semaphore routines, that is, when it has been booted with SYSGEN parameter SYSTEM_CHECK set to 1.

In response to the command SHOW PROCESS/SEMAPHORE, the System Dump Analyzer (SDA) displays the contents of this buffer.

Access to the inner mode semaphore is synchronized through the load-locked/store-conditional mechanism (see Chapter *Synchronization Techniques*).

5.8.1.2 **Acquiring the Inner Mode Semaphore for a System Service.** SYSTEM_SERVICE_DISPATCHER is conditionally compiled. Its listing and object modules are named SYSTEM_SERVICE_DISPATCHER and SYSTEM_SERVICE_DISPATCHER_MON. The latter version maintains the history buffer of inner mode semaphore acquisitions and releases.

EXE$CMODEXEC and EXE$CMODKRNL are the executive and kernel mode system service dispatchers, in module SYSTEM_SERVICE_DISPATCHER. Each tests whether the system service flags (DISP_B_FLAGS) for the requested service have SSFLAG_K_THREAD_SAFE set. If the flag is set, no further action for inner mode multithread synchronization is required. If the flag is clear, each tests whether CTL$GL_MULTITHREAD is greater than 1. If not, no further action is required. If the service is not thread-safe and the process is multithreaded, each dispatcher determines the current kernel thread ID and calls local routine IMSEM_ACQUIRE to acquire the semaphore.

Running in executive or kernel mode, IMSEM_ACQUIRE takes the following steps. (Some of the numbers in the following list correspond to numbers in Figure 5.11, which shows a somewhat simplified example version of the flow.)

① It transforms the kernel thread ID into an IPID and extracts the sequence number.

② It makes the tests described in Section 5.8.1.1 to determine whether the IMSEM$V_CHECK_UPCALL flag should be set or clear in the semaphore once acquired.

③ It executes a load-locked instruction to read the owner and owner depth of the semaphore into a register.

④ If the semaphore is owned by another kernel thread, IMSEM_ACQUIRE first checks whether other kernel threads have already been placed into a wait for the semaphore (IMSEM$V_RS_WAITRS or IMSEM$V_TM_WAITRS is set) and whether the system is a uniprocessor. If either is true, it continues with step 10 in Section 5.8.1.3, possibly placing the kernel thread into a wait.

If neither is true, it spins, testing whether the semaphore is still owned, for 125 microseconds using the process cycle counter (see Chapter *Time Support*). If the semaphore is released during that time, IMSEM_ACQUIRE continues with step 2. If the semaphore is not released, it continues with step 10 in Section 5.8.1.3, possibly placing the kernel thread into a wait.

⑤ If the semaphore is unowned or if it is owned and this kernel thread is the owner (its IPID sequence number matches that of the owner), IMSEM_ACQUIRE increments the owner depth in the register. If necessary, it inserts the thread's sequence number into the register word corresponding to IMSEM$W_OWNER.

⑥ If the semaphore owner depth is 1, IMSEM_ACQUIRE also inserts the desired value for the IMSEM$V_CHECK_UPCALL flag into the register. IMSEM_ACQUIRE then executes a store-conditional instruction to write the register to the semaphore.

If the store fails, it continues with step 3.

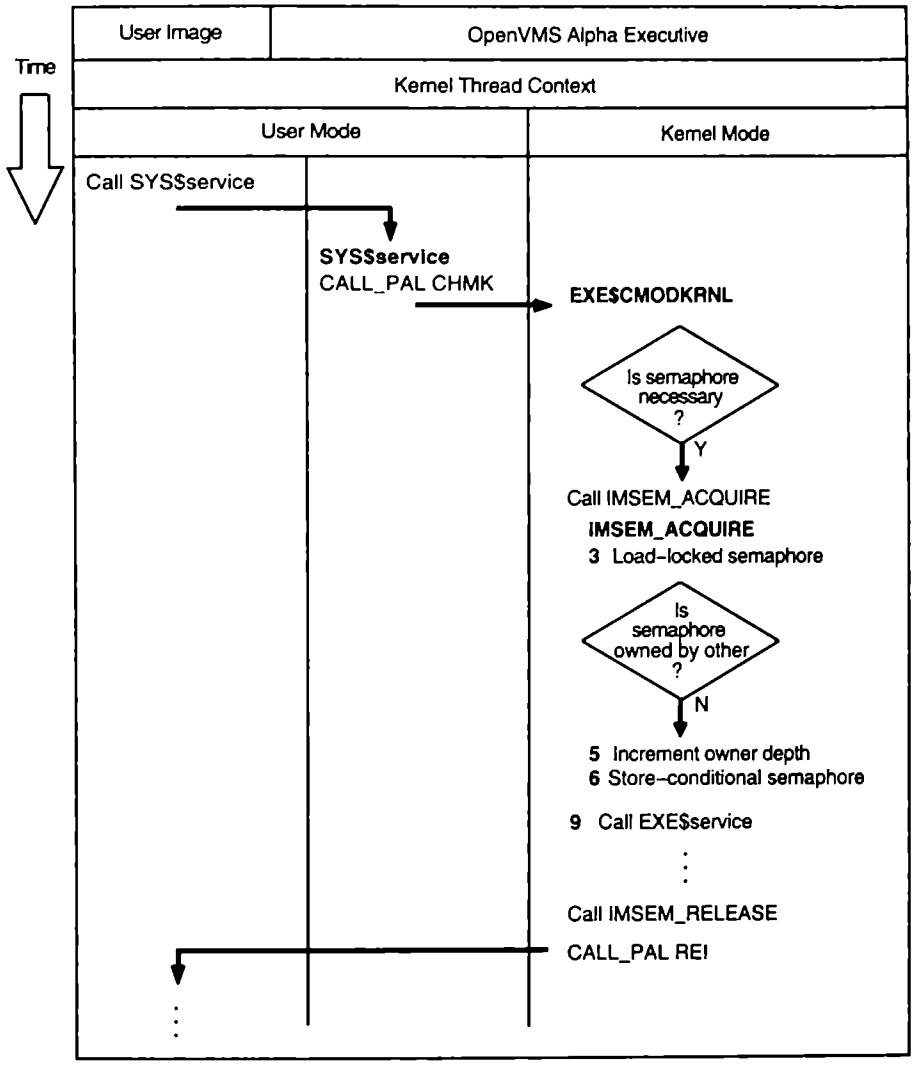



**Figure 5.11**
Example of Acquiring the Inner Mode Semaphore

⑦ If the store succeeds, this kernel thread now owns the semaphore. IMSEM_ACQUIRE executes a memory barrier instruction to order semaphore acquisition with subsequent changes to process data made by the system service.

⑧ The monitoring version of IMSEM_ACQUIRE fills in the next history buffer entry with the code entry address of the system service being requested and the semaphore contents.

⑨ It returns to the system service dispatcher, which continues by calling the requested service.

When the system service procedure returns, the dispatcher releases the semaphore, as described in Section 5.8.1.5.

**5.8.1.3 Waiting for the Inner Mode Semaphore.** If IMSEM_ACQUIRE fails to acquire the semaphore, it continues from the steps in Section 5.8.1.2 to the following steps. (Some of the numbers in the following list correspond to numbers in Figure 5.12. The figure also shows some of the steps in Section 5.8.1.2 that led to this point.

⑩ It tests whether IMSEM$V_CHECK_UPCALL is currently set in the semaphore. If the flag is clear, it continues with step 14.

⑪ If the flag is set, it also checks whether the system service was requested from user mode. If not, it continues with step 14.

⑫ If the flag is set and the service was requested from user mode, IMSEM_ACQUIRE calls EXE$CHECK_UPCALL, in module TM_SUPPORT, to see whether an upcall to a user mode thread manager should be made for an inner mode semaphore wait (see Section 5.9).

⑬ If EXE$CHECK_UPCALL returns a status indicating that an upcall is possible, IMSEM_ACQUIRE sets bit IMSEM$V_TM_WAITRS in the semaphore.

⑭ In any case, it calculates the address of the change mode instruction in the system service transfer routine and modifies the return PC in the change mode exception stack frame to cause the execution of local routine WAIT$JKT.

⑮ It loads the argument, procedure value, and return address registers, restoring them from inner mode stack copies to their values at entry to the system service transfer routine so that the service can be requested again later.

⑯ It executes a `CALL_PAL REI`, using the modified change mode exception frame, and control is transferred to WAIT$JKT.

⑰ WAIT$JKT, running in the mode from which the service was requested, saves the system service argument, procedure value, and return address registers.

⑱ It tests whether an upcall should be made to a user mode thread manager. This test is based on the results of steps 10, 11, and 12.

⑲ If so, it calls EXE$INFORM_TM_CALL, in module TM_SUPPORT, to do so. The user mode thread manager may save the context of the user mode thread and place a different user mode thread into execution in this kernel thread. Section 5.9 provides further information. Whenever execution of this user mode thread is resumed, it continues with step 21.

⑳ If no upcall is possible, WAIT$JKT requests the system service $SS_WAIT_THREAD (see Section 5.8.1.4) to place the kernel thread into a resource wait for the semaphore.

㉑ Later, when the semaphore is available and the kernel thread becomes computable, WAIT$JKT restores the system service argument, procedure value, and return address registers from the outer mode stack and reexecutes the transfer routine for the system service whose requested execution triggered the attempt to acquire the semaphore.

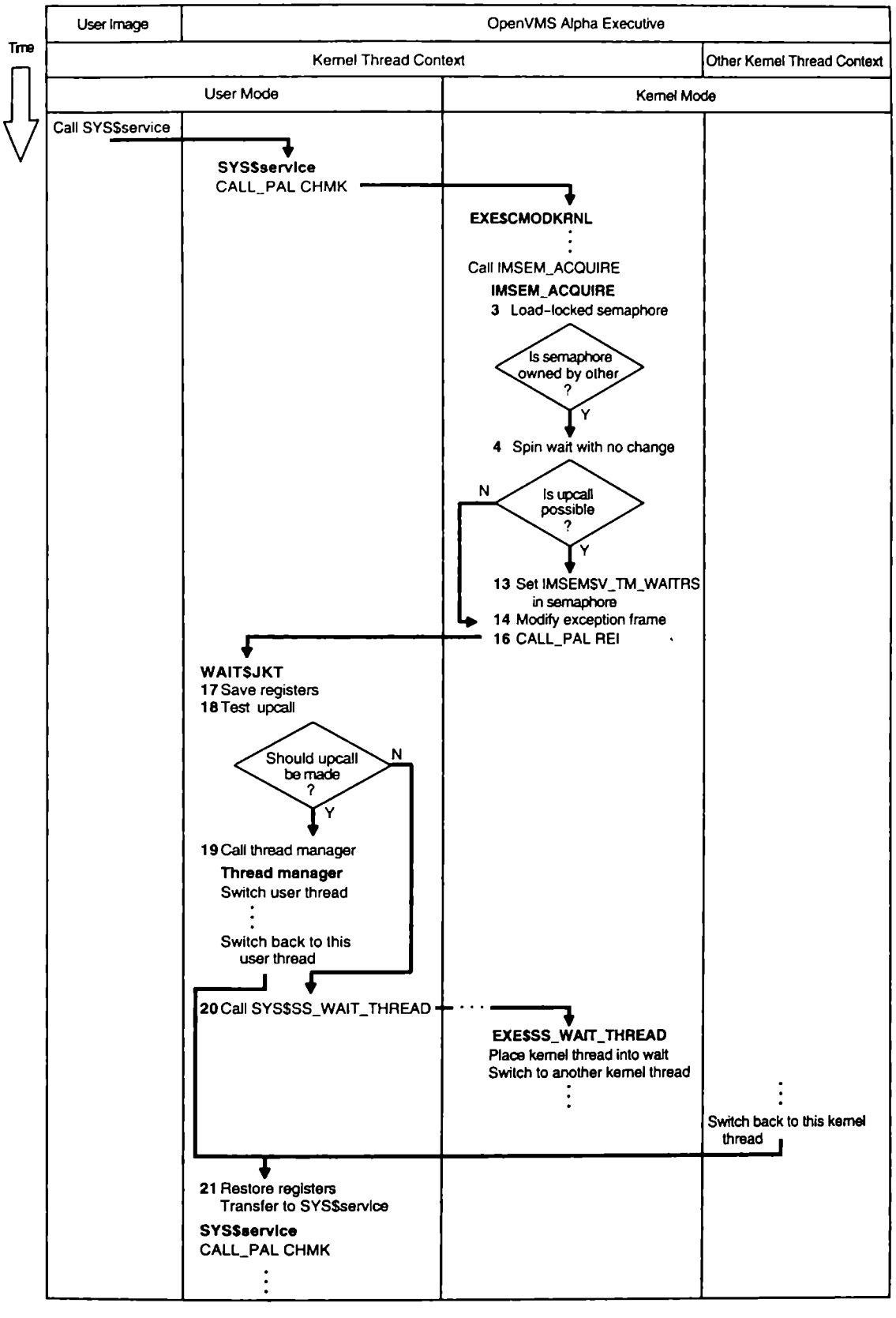



**Figure 5.12**
Example of Waiting for the Inner Mode Semaphore

The change mode dispatcher is reentered and repeats its attempt to acquire the inner mode semaphore on behalf of this kernel thread.

**5.8.1.4 $SS_WAIT_THREAD System Service.** EXE$SS_WAIT_THREAD in module SYSTEM_SERVICE_IMSEM, the $SS_WAIT_THREAD system service procedure, runs in kernel mode. It is requested by routine WAIT$JKT, described in the previous section. It is a thread-safe service and does not require the inner mode semaphore. EXE$SS_WAIT_THREAD takes the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED. The spinlock synchronizes access to the scheduler database, in particular, the MWAIT queue. Acquiring it now enables the combination of setting the IMSEM$V_RS_WAITRS bit and inserting the kernel thread in the MWAIT queue to be atomic.
2. It examines the inner mode semaphore.
3. If the semaphore owner and depth are zero, it releases the spinlock and returns the status SS$_NORMAL. WAIT$JKT will reexecute the system service transfer routine for the original system service, the one whose execution required the inner mode semaphore.
4. If the semaphore is owned, EXE$SS_WAIT_THREAD executes a load-locked instruction preparatory to updating the semaphore flags.
5. If the semaphore has just become unowned, EXE$SS_WAIT_THREAD continues with step 3. Otherwise, it sets the IMSEM$V_RS_WAITRS bit in the semaphore and executes a store-conditional instruction to replace the contents. If the store fails, it continues with step 2.
6. If the store succeeds, it executes a memory barrier instruction.
7. It stores the value of the resource for which the kernel thread will wait in KTB$L_EFWM. The value consists of the kernel thread's process index in the high word and RSN$_INNER_MODE in the low word. This value is specific to the kernel threads in this process.
8. It calls SCH$WAIT_PROC, in module SCHED_ROUTINES, to update the scheduler database by placing the kernel thread into an MWAIT for the inner mode semaphore.
9. Still holding the SCHED spinlock, EXE$SS_WAIT_THREAD returns the status SS$_WAIT_CALLERS_MODE to the system service dispatcher, which calls a scheduling routine to switch the context of this kernel thread with that of another resident kernel thread and to release the spinlock.

The kernel thread waits in the access mode from which it requested the original system service, the one whose execution required the inner mode semaphore. When the semaphore is released, the kernel thread will become computable and reexecute that system service's transfer routine.

Reentered, the system service dispatcher will repeat its attempt to acquire the semaphore.

5.8.1.5 **Releasing the Inner Mode Semaphore.** When a system service procedure returns to the system service dispatcher, EXE$CMODEXEC or EXE$CMODKRNL, in module SYSTEM_SERVICE_DISPATCHER, the dispatcher checks whether the semaphore needs to be released. If the service was not thread-safe and if the process is multithreaded, the dispatcher calls local routine IMSEM_RELEASE to release the semaphore.

Running in executive or kernel mode, IMSEM_RELEASE takes the following steps:

1. The monitoring version fills in the next history buffer entry with the address of its caller (one of several locations in one of the system service dispatchers) and the semaphore contents with the decrement of owner depth represented.
2. It executes a load-locked instruction to get the semaphore contents and lock the structure for a subsequent update.
3. It begins to define new contents for the semaphore in a register. It decrements the owner depth and determines which semaphore flags are set:
   - If the depth is not 0, the new value will be the updated depth and the same owner and flags.
   - If the depth is 0, if no inner mode ASTs are pending, and if there are no resource waiters, the new value for the semaphore will be 0.
   - If the depth is 0 and there are pending inner mode ASTs, the new value will be a depth of 0 and the current owner field and AST and IMSEM$V_RS_WAITRS bits left intact. The other bits are cleared. That is, this kernel thread continues to own the semaphore. One or more pending inner mode ASTs will be delivered to it before it releases the semaphore.
4. If the depth is now 0, if no inner mode ASTs are pending, and if there are resource waiters, IMSEM_RELEASE requests the system service $SS_RELEASE_THREAD (see Section 5.8.1.7) to release the semaphore and make computable any kernel threads waiting for this semaphore. The service will establish a new value for the semaphore. When it returns, IMSEM_RELEASE continues with step 6.
5. If the depth is not yet 0, or if it is 0 and inner mode ASTs are still pending, IMSEM_RELEASE cannot release the semaphore. It executes a store-conditional instruction to update the semaphore with the contents determined at step 3. If the store fails, IMSEM_RELEASE continues with step 2. If the store succeeds, it executes a memory barrier to force ordering of changes to process data with the release of the semaphore.

6. If the owner depth was decremented to 0 and bit IMSEM$V_TM_WAITRS was set in the semaphore before it was released, IMSEM_RELEASE continues with the actions described in Section 5.8.1.6, which notify the user mode thread manager.
7. IMSEM_RELEASE returns to its caller, the system service dispatcher, which returns to the system service requestor.

5.8.1.6 **Releasing User Mode Threads.** If IMSEM_RELEASE finds that the user mode thread manager must be notified of the semaphore release, it takes the following steps:

1. It calculates the address of the change mode instruction in the system service transfer routine and modifies the return PC in the change mode exception stack frame to cause the execution of local routine RELEASE$JKT.
2. It executes a `CALL_PAL REI`, using the modified change mode exception frame, and control is transferred to RELEASE$JKT.
3. RELEASE$JKT, running in user mode, calls EXE$CHECK_UPCALL, in module TM_SUPPORT, to see whether an upcall to a user mode thread manager should be made for an inner mode semaphore release.
4. If EXE$CHECK_UPCALL returns a status indicating that an upcall is possible, RELEASE$JKT calls EXE$INFORM_TM_CALL, in module TM_SUPPORT.

   The user mode thread manager may save the context of the user mode thread and place a different user mode thread into execution in this kernel thread. Section 5.10.5 provides further information. Whenever execution of this user mode thread is resumed, it continues with step 5.
5. In either case, RELEASE$JKT returns to the requestor of the system service whose request triggered the semaphore acquisition and whose exit triggered its release.

5.8.1.7 **$SS_RELEASE_THREAD System Service.** EXE$SS_RELEASE_THREAD in module SYSTEM_SERVICE_IMSEM, the $SS_RELEASE_THREAD system service procedure, runs in kernel mode. It is requested by routine IMSEM_RELEASE, described in Section 5.8.1.5. It is also called from AST delivery code after one or more ASTs requiring the inner mode semaphore have been delivered to a kernel thread and the semaphore is being released. $SS_RELEASE_THREAD is a thread-safe service and does not require the inner mode semaphore.

EXE$SS_RELEASE_THREAD takes the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
2. It executes a load-locked instruction to read the inner mode semaphore contents.

3. It determines a new value for the semaphore based on the state of the AST pending bits:
   - If any is set, the new value will be a depth of 0 and the current owner field and AST pending bits.
   - Otherwise, the new value will be 0.
4. It executes a store-conditional instruction to update the semaphore with its new contents. If the store fails, EXE$SS_RELEASE_THREAD continues with step 2. If the store succeeds, it executes a memory barrier instruction.
5. It constructs the value that represents the inner mode semaphore resource for which kernel threads in this process are waiting. The value consists of the kernel thread's process index in the high word and RSN$_INNER_MODE in the low word.
6. It calls EXE$WAIT_QUEUE_RELEASE, in module MUTEX, to make computable all kernel threads waiting for that resource, namely, all kernel threads in this process waiting for the inner mode semaphore.
7. It releases the SCHED spinlock and returns the state of IMSEM$V_TM_WAITRS read in step 2.

### 5.8.2 ASTs

In a single-threaded process ASTs of a given access mode are delivered in the order in which they were queued. ASTs of an inner mode are delivered before any outer mode ASTs. In a multithreaded process, however, there is no implicit serialization of mode. An AST is not necessarily delivered to the specific kernel thread that requested it:

- In a process with a user mode thread manager, each user mode AST is transformed into an upcall to the thread manager, which is responsible for the actual delivery.
- A supervisor mode AST is generally queued to the kernel thread that requested the system service associated with the AST.
- A thread-safe executive or kernel mode AST is queued and delivered to the kernel thread that requested it.
- A non-thread-safe executive or kernel mode AST is delivered to whichever kernel thread currently owns the inner mode semaphore, if any. If the semaphore is unowned, the semaphore is acquired on behalf of the specific kernel thread to which the AST is targeted, and the AST delivered to that thread.

Section 5.8.2.1 summarizes changes to AST-related data structures required for multithreading support. Section 5.8.2.2 describes queuing an AST to a multithreaded process. Section 5.8.2.3 summarizes AST delivery in a multithreaded process.

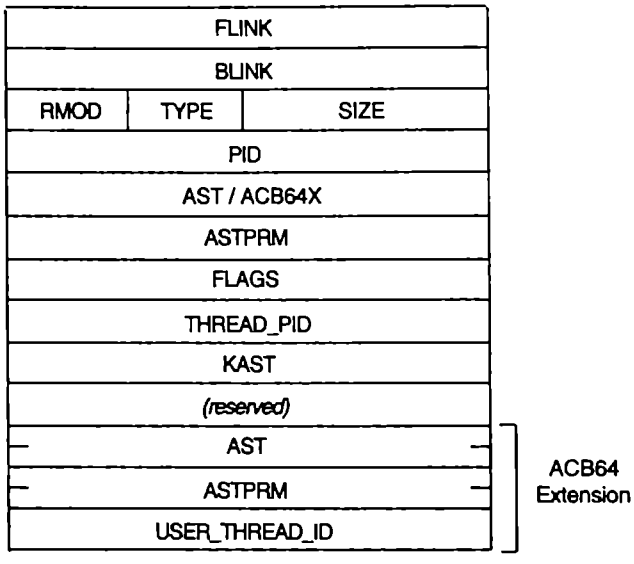



**Figure 5.13**
Multithreading Support in the AST Control Block (ACB)

5.8.2.1 **AST-Related Data Structures.** The multithreading support provided by OpenVMS Alpha Version 7.0 required several changes to the ACB structure from previous versions. Figure 5.13 shows the present layout of the ACB.

The fields with particular relevance to multithreading support are the following:

- ACB$B_RMOD bit ACB$V_FLAGS_VALID, when set, means that the ACB includes a valid ACB$L_FLAGS field.
- ACB$L_PID can contain the IPID of the process or of the kernel thread for which the AST is intended.
- ACB$L_FLAGS defines several bits related to multithreading:
  - Bit ACB$V_THREAD_SAFE, when set, means that this ACB represents an inner mode AST that does not require the inner mode semaphore to run.

    If ACB$V_FLAGS_VALID is clear, the ACB is presumed to be non-thread-safe.

    Note that if the ACB also describes a piggyback special kernel mode AST, the state of this bit applies to the piggyback AST routine as well as to any inner mode normal AST described by the ACB.
  - Bit ACB$V_THREAD_PID_VALID, when set, means that the contents of ACB$L_THREAD_PID are valid. The Enqueue Lock Request ($ENQ) system service, for example, in queuing a blocking AST, sets this bit and initializes ACB$L_THREAD_PID with the IPID of the kernel thread that currently owns the lock associated with the AST.
  - Bit ACB$V_UPCALL, when set, means that this ACB represents an up-call to a user mode thread manager.

- Bit ACB$V_64BITS, when set, means that the ACB contains an extension. Originally defined for 64-bit address support, the extension includes a field specific to multithreading support. ACB$L_AST contains the offset to the extension from the beginning of the ACB.
- Bit ACB$V_USER_THREAD_ID_VALID, when set, means that ACB64$L_USER_THREAD_ID contains a valid user mode thread identification.

◆ ACB$L_THREAD_PID contains the IPID of the kernel thread for which the AST is intended.

◆ ACB64$L_USER_THREAD_ID is intended to identify the user thread for which a user mode AST is intended. In OpenVMS Alpha Version 7.0, it always contains the value 0.

Each kernel thread has five AST listheads in its KTB, one for each access mode plus one for special kernel mode ASTs (*x* stands for F or B—forward or backward—link): KTB$L_ASTQ*x*L_SPK, KTB$L_ASTQ*x*L_K, KTB$L_ASTQ*x*L_E, KTB$L_ASTQ*x*L_S, and KTB$L_ASTQ*x*L_U.

The condition (empty or not) of each of these queues is summarized by one of the low-order four bits in KTB$L_AST_PENDING. If the queue contains one or more ACBs, the bit is set. Bit 0 represents both the special kernel mode and kernel mode queues.

In addition, the PCB contains processwide listheads for executive mode, kernel mode, and special kernel mode ASTs that require the inner mode semaphore: PCB$L_IM_ASTQ*x*L_E, PCB$L_IM_ASTQ*x*L_K, and PCB$L_IM_ASTQ*x*L_SPK.

The condition (empty or not) of these queues is summarized by the inner mode semaphore flags IMSEM$V_KAST_PENDING, which reflects the special kernel mode and kernel mode queues, and IMSEM$V_EAST_PENDING, which pertains to executive mode queues.

Bit PCB$V_RWAST in PCB$L_STS2 is set to indicate that one or more kernel threads in the process are in resource wait mode waiting for RSN$_ASTWAIT.

5.8.2.2 **Queuing ASTs to a Multithreaded Process.** All executive routines queue an AST by calling SCH$QAST, in module ASTDEL. SCH$QAST takes the following steps to queue an AST to a multithreaded process:

1. It acquires the SCHED spinlock.
2. It gets the address of the PCB corresponding to the IPID in ACB$L_PID, confirms that the process exists, and determines that it has more than one kernel thread.
3. SCH$QAST determines which kernel thread is the requested target of this AST. If ACB$V_FLAGS_VALID in ACB$B_RMOD and ACB$V_THREAD_PID_VALID in ACB$L_FLAGS are both set, SCH$QAST gets

the IPID of that kernel thread from ACB$L_THREAD_PID. Otherwise, it uses the contents of ACB$L_PID.

4. It gets the address of the corresponding KTB. If that kernel thread has been deleted, SCH$QAST gets the address of the initial kernel thread's KTB.
5. If the ACB requires the inner mode semaphore, SCH$QAST inserts the ACB at the end of one of the three processwide PCB queues, depending on its mode and type.
6. If the ACB does not require the semaphore or is an ACB for an outer mode, SCH$QAST inserts it at the end of one of the five KTB queues, depending on its mode and type.
7. It tests whether the kernel thread's process is resident or outswapped.
   - If the process is outswapped, SCH$QAST tests whether the AST requires the inner mode semaphore. If not, it sets the bit in KTB$L_AST_PENDING corresponding to the mode of the AST. SCH$QAST reports an AST queuing event for the kernel thread, releases the SCHED spinlock, and returns to its caller. (If the process is outswapped, the inner mode semaphore is inaccessible.)
   - If the process is resident, SCH$QAST tests whether the AST requires the semaphore. If not, it sets the bit in KTB$L_AST_PENDING and continues with step 11.
8. If the AST requires the inner mode semaphore, SCH$QAST executes a load-locked instruction to read the semaphore, using the system space mapping of the PHD. It determines whether the semaphore is free and, if not, which kernel thread owns it.
9. It executes a store-conditional instruction to write the semaphore with new contents, based partly on the old contents but with a bit set corresponding to the mode of the AST:
   - If the semaphore was unowned, IMSEM$W_OWNER gets the sequence number part of the IPID of the kernel thread for which this ACB is being queued. That kernel thread now owns the semaphore.
   - If the semaphore was owned, IMSEM$W_OWNER retains its original contents.

   SCH$QAST executes a memory barrier instruction.
10. If the semaphore was owned, SCH$QAST determines the address of the KTB of its owner. This is the kernel thread that will take the AST delivery interrupt.
11. Whether or not the AST is thread-safe, SCH$QAST calculates appropriate values for the target kernel thread's AST summary register and stores them in that register or the kernel thread's FRED, as appropriate. Chapter *ASTs* has further details.
12. It releases the SCHED spinlock and returns to its caller.

Later, an AST delivery interrupt is triggered in the target kernel thread.

5.8.2.3 **AST Delivery in a Multithreaded Process.** OpenVMS Alpha has four AST delivery interrupts, one for each mode. Multithreading support affects the delivery of kernel, executive, and user mode ASTs. Delivery of supervisor mode ASTs is not affected. The following issues are relevant to inner mode AST delivery:

- The kernel thread that owns the inner mode semaphore must service the processwide non-thread-safe AST queues and clear each queue state bit in the inner mode semaphore when appropriate. In processing the non-thread-safe AST queues, it must maintain AST access mode ordering and process thread-safe queues as well.
- In servicing the processwide queues, the kernel thread may take delivery of an AST intended for a second kernel thread, the queuing of which would have changed the second kernel thread's scheduling state to computable. Consequently, after the AST routine executes, an AST event must be reported to OpenVMS scheduling code on behalf of the second kernel thread to change its state.
- After all non-thread-safe AST queues have been serviced, executive code must release ownership of the inner mode semaphore. If the semaphore depth goes to 0, any kernel threads in the same process waiting for the semaphore must be made computable.
- The monitoring version of AST delivery and exit code must fill in inner mode semaphore history buffer entries.
- A piggyback special kernel mode AST procedure must be called with the inner mode semaphore acquired unless the ACB$V_THREAD_SAFE bit is set.

The kernel mode AST delivery routine SCH$ASTDEL_K, in module ASTDEL, deals with special kernel mode ASTs and kernel mode ASTs, both those that require the inner mode semaphore and those that do not. Running in a multithreaded process's kernel thread, it must read the inner mode semaphore to determine whether a non-thread-safe kernel mode AST is pending and, if so, whether the current kernel thread owns the inner mode semaphore.

If both are true, SCH$ASTDEL_K services each ACB in the PCB$L_IM_ASTQFL_SPK queue. After the special kernel mode AST procedure returns, SCH$ASTDEL_K checks whether it was intended for a different kernel thread but delivered to this one because of the latter's ownership of the inner mode semaphore. If so, SCH$ASTDEL_K reports an AST queuing scheduling event for the intended kernel thread (see Chapter 2). If the intended kernel thread would have been made computable had the ACB been queued to it, this event report will serve the same purpose.

After servicing all non-thread-safe special kernel mode ASTs, SCH$ASTDEL_K services all thread-safe special kernel mode ASTs queued to this kernel thread.

It then services one non-thread-safe kernel mode AST in the PCB$L_IM_ASTQFL_K queue. The AST procedure returns to AST exit code. If PCB$V_RWAST was set in PCB$L_STS2, the exit code reports the resource RSN$_ASTWAIT available for all kernel threads of this process waiting for that resource. Any such kernel threads are made computable.

If the ACB was intended for another kernel thread, AST exit code reports an AST queuing scheduling event for the intended kernel thread. The AST exit code then tries to remove the next ACB from the PCB$L_IM_ASTQFL_K queue. If one was removed, it calls SCH$ASTDEL_K to deliver it. This sequence is repeated until the queue is empty.

After removing the last non-thread-safe kernel mode AST, SCH$ASTDEL_K clears bit IMSEM$V_KAST_PENDING in the inner mode semaphore and services that AST, reporting an AST queuing event if necessary. The AST exit code decrements the semaphore owner depth. If the depth is now 0, it tests IMSEM$V_EAST_PENDING:

- If IMSEM$V_EAST_PENDING is set, the AST exit code does not release the semaphore: some pending executive mode ASTs require the inner mode semaphore and must be serviced first.
- Otherwise, if IMSEM$V_EAST_PENDING is clear, the AST exit code releases the semaphore. If bit IMSEM$V_RS_WAITRS was set, other kernel threads in this process have been waiting to acquire the semaphore. The AST exit code calls EXESSS_RELEASE_THREAD (see Section 5.8.1.7) to make them computable.

In either case, SCH$ASTDEL_K then services ACBs in the thread-safe kernel mode queue. When the queue is empty, it dismisses the interrupt. When access mode is changed from kernel, if this kernel thread still owns the semaphore and if non-thread-safe executive mode ASTs are pending, there will be another AST interrupt for delivery of the executive mode ASTs.

The sequence for delivering an executive mode AST to a kernel thread resembles that for delivering a normal kernel mode AST, except that only one queue needs to be serviced and, after returning to executive mode to deliver an executive mode AST, the executive needs to run in kernel mode to execute the AST exit code. SCH$ASTDEL, in module ASTDEL, the AST delivery interrupt routine for mode other than kernel, reads the inner mode semaphore to test whether an executive mode AST is pending that requires the semaphore and, if so, whether this kernel thread owns the semaphore. If both are true, it processes the first ACB in the queue at PCB$L_IM_ASTQFL_E. The processing resembles that for other outer mode ASTs: registers must be saved and a `CALL_PAL REI` instruction must be executed to change mode from kernel to executive mode.

When the AST procedure returns, AST exit code changes to kernel mode. If PCB$V_RWAST was set in PCB$L_STS2, AST exit code reports the resource RSN$_ASTWAIT available for all kernel threads of this process waiting for

that resource. It also reports an AST queuing scheduling event for the original target of the AST, if it was not this kernel thread. The AST exit code then tries to remove the next ACB from the PCB$L_IM_ASTQFL_E. If one was removed, it calls SCH$ASTDEL to deliver it. This sequence is repeated until the queue is empty.

When the queue is empty, IMSEM$V_EAST_PENDING is cleared and the semaphore owner depth decremented. If the semaphore is now free and bit IMSEM$V_RS_WAITRS was set, the AST exit code calls EXE$SS_RELEASE_THREAD and then services any pending thread-safe executive mode ASTs before dismissing the AST interrupt.

When a user mode AST is delivered to a kernel thread, SCH$ASTDEL tests whether ACB$V_UPCALL is set. If so, it simply delivers the user mode AST in the usual way (see Chapter *ASTs*). Otherwise, it tests whether a user mode thread manager has registered a user mode AST callback routine. If so, SCH$ASTDEL copies the AST procedure address and parameter to the current kernel thread's PKTA$Q_AST_ADDRESS and PKTA$Q_AST_PARAMETER. After having executed a `CALL_PAL REI` to return to user mode, SCH$ASTDEL calls EXE$UAST_AST, in module TM_SUPPORT, passing it the address of PKTA$Q_AST_ADDRESS. EXE$UAST_AST performs a callback to the user mode thread manager (see Section 5.10.6).

When an outer mode AST with a piggyback special kernel mode AST is delivered to a kernel thread, the AST delivery interrupt service routine checks whether the piggyback AST is thread-safe. If not, it must acquire the inner mode semaphore. If the semaphore is owned by another kernel thread, the interrupt service routine takes steps similar to those described in Section 5.8.1.3 to place the kernel thread into a wait for the inner mode semaphore.

### 5.8.3 $SINGLE_THREAD System Service

The $SINGLE_THREAD system service is used for debug breakpoint handling and the Digital command language's CTRL/Y handling. It is requested from one kernel thread in a multithreaded process to force all the other kernel threads in that process to cease any execution in a particular access mode and any outer modes. All the other kernel threads remain quiescent until the initiating kernel thread requests $SINGLE_THREAD to reenable normal operation. The quiescent kernel threads wait in the SUSP scheduling state.

If the service was requested from user or supervisor mode, the quiescent kernel threads are put into a state suspension in which they can receive inner mode ASTs. If the service was requested from an inner mode, the quiescent kernel threads are not made computable for AST delivery.

The DECthreads thread manager first tries the $FORCE_QUANTUM_END system service, described in Section 5.9.2, to break into the execution

of currently running user mode threads. If that fails, it requests the $SINGLE_THREAD system service.

EXE$SINGLE_THREAD in module TM_SUPPORT, the $SINGLE_THREAD system service procedure, runs in kernel mode. Its input arguments specify the nature of the request. The ENABLE_FLAG argument, when set, specifies that single threading is being initiated. When clear, it specifies that single threading is to be terminated. The ACCESS_MODE argument specifies the mode at which other kernel threads should be blocked.

EXE$SINGLE_THREAD takes the following steps:

1. It confirms that the output argument to receive the previous state of single threading specifies a writable address and, if not, returns the error status SS$_ACCVIO.
2. It checks whether the process currently has more than one kernel thread and, if not, returns the status SS$_NORMAL.
3. It maximizes the ACCESS_MODE argument with the mode from which the service was requested to get the mode at which the other threads should be blocked.
4. It examines the PCB$V_SINGLE_THREADED bit in PCB$L_STS2 that corresponds to the mode associated with the single-thread request to see if the process is already in that state. If so, it returns SS$_NORMAL.
5. It determines the thread ID of the current thread and the address of its KTB.
6. If the ENABLE_FLAG argument specifies that single threading should be terminated, EXE$SINGLE_THREAD continues with step 13.
7. If the ENABLE_FLAG argument specifies that single threading should be enabled, EXE$SINGLE_THREAD raises IPL to 2. It scans the process's KTB vector. For each kernel thread it finds other than itself, it allocates and clears nonpaged pool for use as an ACB and records the ACB address.

   On a failure to allocate an ACB, it deallocates all those previously allocated and returns the failure status to its requestor.
8. EXE$SINGLE_THREAD acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
9. It sets the PCB$V_SINGLE_THREADED bit corresponding to the mode from which the service was requested. It also stores the current kernel thread's ID in the PCB$Q_ST_KT_ARRAY word corresponding to that access mode to indicate that this kernel thread has requested single threading.
10. It examines each KTB in the process to determine whether a single-thread request at this mode is already active.
    - If so, it deallocates the ACB.
    - If not, it initializes the ACB to describe an AST for the target kernel thread with AST procedure EXE$SINGLE_THREAD_AST and AST

argument equal to the mode from which the $SINGLE_THREAD service was requested. The AST is a thread-safe one. Its access mode is 1 less than the value of the ACCESS_MODE argument. Requesting that the process become single-threaded in user mode results in queuing a supervisor mode AST to each kernel thread other than the requestor. Requesting that the process become single-threaded in kernel mode results in queuing a kernel mode AST. EXE$SINGLE_THREAD queues the AST.

11. If the service was requested from user or supervisor mode, EXE$SINGLE_THREAD sets PCB$V_SOFT_SINGLE_THREAD in the PCB to enable AST delivery to the kernel threads that will suspend themselves in response to the queued ASTs. It returns SS$_NORMAL to its requestor.
12. If the service was requested from executive or kernel mode, EXE$SINGLE_THREAD stores the number of other kernel threads in PCB$L_ST_ACK_COUNT and releases the SCHED spinlock to give each of the other kernel threads an opportunity to be scheduled, receive the AST, and acknowledge the single-thread request by decrementing PCB$L_ST_ACK_COUNT.

    EXE$SINGLE_THREAD immediately reacquires the SCHED spinlock and tests PCB$L_ST_ACK_COUNT. If the field is nonzero, some kernel threads have not yet acknowledged the request, and it hibernates the current kernel thread, releasing the spinlock. Each time the hibernating thread awakens, EXE$SINGLE_THREAD repeats the test. When the field goes to zero, the routine returns SS$_NORMAL to its requestor.
13. If single threading is being terminated in the process, EXE$SINGLE_THREAD clears the bit corresponding to the mode from which the service was requested.
14. It clears the bit corresponding to the kernel thread's ID in PCB$Q_ST_KT_ARRAY to indicate that the thread is ending its request for the process to be single-threaded.
15. It sets or clears PCB$V_SOFT_SINGLE_THREAD based on whether the kernel or executive mode PCB$V_SINGLE_THREADED bit is set.
16. It scans the process's KTB vector and reports a resume event for each other kernel thread in the process.
17. It releases the SCHED spinlock and returns to its requestor.

The AST procedure EXE$SINGLE_THREAD_AST runs in each kernel thread in the specified mode. It requests the $SINGLE_THREAD_INT system service to place the current kernel thread into a state of suspension.

The $SINGLE_THREAD_INT system service procedure, EXE$SINGLE_THREAD_INT in module TM_SUPPORT, sets the bit corresponding to the mode from which the $SINGLE_THREAD system service was requested in KTB$V_SINGLE_THREAD_ACT. If that mode is executive or kernel, EXE$SINGLE_THREAD_INT also decrements PCB$L_ST_ACK_COUNT. If

it decrements the count to zero, it awakens the kernel thread that requested the $SINGLE_THREAD system service. It clears the AST active bit corresponding to the mode of the AST procedure and places the kernel thread into the suspended state.

## 5.9 UPCALLS FROM THE OPENVMS EXECUTIVE TO A USER MODE THREAD MANAGER

The term *upcall* refers to a call from an executive routine to a routine in a user mode run-time library. Such a call must be made in user mode and therefore is often preceded by an access mode change to the outer, less privileged mode. Sometimes, however, the executive has already changed to user mode preparatory to placing a kernel thread into a wait; in this case, it simply calls the user mode thread manager.

The OpenVMS executive performs an upcall to notify a user mode thread manager of a state change in a kernel thread. The state change might be one that stalls the current user mode thread, or it might make a previously stalled user mode thread executable. Upcalls also signal the thread manager of some action it must take, for example, delivery of an AST to a user mode thread.

Events that can lead to upcalls are the following:

- $SYNCH system service requested for an operation not yet complete
- Wait for Logical OR of Event Flags ($WFLOR), Wait for Logical AND of Event Flags ($WFLAND), or Wait for Single Event Flag ($WAITFR) system service requested and flags are not set
- Hibernate ($HIBER) system service requested
- Page fault wait or page fault complete
- Inner mode semaphore wait or release
- Event flag setting event
- Wake event
- $SETAST system service requested
- Delivery of user mode AST
- User mode thread quantum expiration
- Exit ($EXIT) or Force Exit ($FORCEX) system service requested

When a user mode thread manager registers itself, it specifies the address of an upcall table containing a list of routines to be called in response to upcall events. The DECthreads thread manager specifies routines for all possible upcall events.

### 5.9.1 Executive Upcalls

An OpenVMS executive routine that might make an upcall first checks whether one is possible by calling EXE$CHECK_UPCALL, in module TM_SUPPORT, or by invoking the macro CHECK_UPCALL.

All the following conditions must be met for an upcall:

- The process must be multithreaded and must have established a user mode thread manager (nonzero contents of CTL$GQ_TM_CALLBACKS).
- The thread manager must have an entry for upcalls of that type.
- The service must have been called from code running in user mode.
- For a blocking upcall, the user mode thread manager must not already be active in this execution context.
- There must be a nonzero value in the hardware process unique value (a field within the HWPCB). The value is expected to be the address of a thread environment block (see Section 5.10.2).

If an upcall is not possible (if any of the previous conditions is not met), the executive routine simply takes whatever action it would have in a process without a user mode thread manager.

If an upcall is possible, the executive routine calls either EXE$INFORM_TM_CALL or EXE$INFORM_TM_AST, both in module TM_SUPPORT. EXE$INFORM_TM_CALL is called from a routine already running in user mode (see Section 5.9.1.1). Called from a routine running in an inner mode, EXE$INFORM_TM_AST queues a user mode AST to call the user mode thread manager (see Section 5.9.1.2).

The following executive routines call EXE$INFORM_TM_CALL.

SCH$PAGEFAULT_UPCALL_JKT, in module SCHEDULER, reports a page fault or collided page fault wait. It passes a flag representing the type of fault and the beginning virtual address of the page or cluster of pages being faulted in. The DECthreads thread manager blocks the user mode thread, saves its context, records information about the wait, and switches to another user mode thread. Control cannot return to SCH$PAGEFAULT_UPCALL_JKT until DECthreads reschedules the blocked user mode thread, typically in response to a page fault complete upcall. When control does return to SCH$PAGEFAULT_UPCALL_JKT, the service restores saved registers and executes a `CALL_PAL REI` instruction to return to the original user mode thread at the instruction that incurred the page fault.

EXE$SETAST, in module SYSASTCON, reports that the $SETAST system service has been requested from user mode. It passes the system service argument, a value specifying whether delivery is being enabled or disabled. Section 5.10.6 describes the DECthreads thread manager's response.

EXE$EXIT, in module SYSEXIT, a mode of caller service, tries to perform an upcall to report that user mode code has requested the $EXIT system service. Section 5.10.7 describes the DECthreads thread manager's response. If EXE$EXIT is unable to perform the upcall because PKTA$V_TM_ACTIVE is already set, it clears PKTA$V_TM_ACTIVE and requests the internal $EXIT_INT service on the presumption that an upcall has already occurred and that DECthreads itself is now requesting the $EXIT service.

EXE$EXIT_INT is responsible for scanning the lists of declared exit handlers and calling them (see Chapter 2). Clearing PKTA$V_TM_ACTIVE makes possible any upcalls resulting from subsequent actions of user mode exit handlers.

EXE$HIBER, in module SYSPCNTRL, reports that user mode code has requested the $HIBER system service. The DECthreads thread manager blocks the user mode thread, saves its context, and switches to another user mode thread. Control cannot return to EXE$HIBER until DECthreads reschedules the blocked user mode thread, typically in response to a wake event. When control does return to EXE$HIBER, the service returns to its requestor, the original user mode thread.

EXE_STD$SYNCH_LOOP, part of the $SYNCH system service, in module SYSSYNCH, reports that user mode code is waiting for a local event flag or for EFN$C_ENF (see Chapter 3) to be set. The DECthreads thread manager blocks the user mode thread, records for which flag the thread waits, and switches to another user mode thread. Control cannot return to EXE_STD$SYNCH_LOOP until DECthreads reschedules the blocked user mode thread, typically in response to an event flag setting event. When control does return to EXE_STD$SYNCH_LOOP, the routine returns to its requestor, the original user mode thread.

EXE$WFLAND and EXE$WFLOR, in module SYSWAIT, report that user mode code is waiting for flags in a local event flag cluster. The DECthreads thread manager blocks the user mode thread, records for which flags the thread waits, and switches to another user mode thread. Control cannot return to the wait service until DECthreads reschedules the blocked user mode thread, typically in response to an event flag setting event. When control does return to the wait service, it returns to its requestor, the original user mode thread.

EXE$WAITFR, in module SYSWAIT, reports that user mode code is waiting for a local event flag. The DECthreads thread manager blocks the user mode thread, records for which flag the thread waits, and switches to another user mode thread. Control cannot return to EXE$WAITFR until DECthreads reschedules the blocked user mode thread, typically in response to an event flag setting event. When control does return to EXE$WAITFR, the routine returns to its requestor, the original user mode thread.

The following executive routines call EXE$INFORM_TM_AST.

Local routine PAGIO_DONE, in module IOCIOPOST, when page fault I/O completes, reformats the I/O request packet (IRP) into an ACB and reports a page fault completion event to the thread manager. The AST is delivered to the kernel thread in whose context the page fault occurred. The AST parameter is the starting virtual address of the page or pages that were faulted in ORed with the number of pages that were read.

SCH$WAKE, in module RSE, reports a WAKE event. The AST is delivered to the kernel thread that is the target of the wake request.

EXE$FORCEX, in module SYSFORCEX, reports a forced exit request. The AST is delivered to the kernel thread that is the target of the exit request.

Three other types of events are reported to a user mode thread manager as AST upcalls:

- User mode thread quantum end, described in Section 5.9.2
- Event flag setting event, described in Section 5.9.3
- User mode AST delivery, described in Section 5.8.2.3

5.9.1.1 **Actions of EXE$INFORM_TM_CALL.** EXE$INFORM_TM_CALL runs in the context of the kernel thread about whose user mode thread the thread manager is being notified. It is called with an argument specifying the type of upcall and zero or more upcall-type-specific arguments. Running typically in user mode, it takes the following steps:

1. It checks that the current access mode is user, generating the fatal bugcheck INCON_SCHED if not. The effect of the bugcheck depends on the current mode: if the process is running in kernel or executive mode, the system will crash; otherwise, the current image will be exited.
2. EXE$INFORM_TM_CALL disables user mode ASTs by executing a `CALL_PAL SWASTEN` instruction. This prevents any upcalls to the thread manager that could lead to its context-switching this user mode thread. Accessing the correct PKTA in the next several steps requires that user mode thread context switches be blocked.
3. It executes the READ_UNQ PALcode routine to get the address of the current user mode thread's thread environment block.
4. It examines TEB$PQ_PER_KT_AREA to get the address of the per kernel thread area for the current kernel thread. If the field contains zero, EXE$INFORM_TM_CALL requests the $GET_PER_KT_AREA system service to get the address.
5. It sets PKTA$V_TM_ACTIVE in PKTA$Q_FLAGS unless the bit is already set.
6. If ASTs had been previously enabled, it reenables them.
7. It calls the thread manager routine specific to this type of upcall, which is typically but not always a blocking upcall.

   At this point, the thread manager may save the context of this user mode thread and switch a different user mode thread into this kernel thread. If so, control does not return here until the original user mode thread is unblocked and placed back into execution. At that point, the thread manager will no longer be active.
8. When control returns to EXE$INFORM_TM_CALL, it tests whether PKTA$V_TM_ACTIVE was clear when it was first entered to make the upcall and, if so, temporarily disables user mode AST delivery while it clears PKTA$V_TM_ACTIVE.
9. It returns to its caller.

5.9.1.2 **Actions of EXE$INFORM_TM_AST.** EXE$INFORM_TM_AST is called with arguments specifying the type of upcall, an AST argument whose meaning and value are specific to the type of upcall, the IPID of the target kernel thread, and optionally the address of an already initialized ACB. Running in kernel mode, it takes the following steps:

1. It gets the address of the KTB associated with the target kernel thread.
2. It determines the address of a jacket AST procedure based on the upcall type.
3. If no ACB was passed to it, it allocates one from nonpaged pool and initializes it to describe a user mode AST. It sets the ACB$V_UPCALL bit in ACB$L_FLAGS to indicate that the AST represents an upcall, preventing the generation of yet another upcall.
4. EXE$INFORM_TM_AST stores the passed AST argument and the address of the jacket procedure in the ACB.
5. It queues the AST to the target kernel thread.
6. It returns to its caller.

The jacket AST procedures are in module TM_SUPPORT. Running as user mode AST procedures, each calls EXE$CALLBACK_JACKET, in module TM_SUPPORT, passing it the upcall type and AST parameter, whose meaning varies with upcall type.

EXE$CALLBACK_JACKET takes the following steps:

1. It confirms that the current mode is user, generating the fatal bugcheck INCON_SCHED if not.
2. It executes the PALcode routine READ_UNQ to get the value of the hardware process unique value, the address of the user mode thread's thread environment block.
   - If the value is zero, EXE$CALLBACK_JACKET requests the $GET_PER_KT_AREA system service to get the address of the PKTA.
   - If the value is nonzero, EXE$CALLBACK_JACKET gets the address of the PKTA from TEB$PQ_PER_KT_AREA.
3. It sets PKTA$V_TM_ACTIVE. Because it is running as an AST procedure, it does not need to disable AST delivery to block user mode thread switching while it accesses the PKTA.
4. It calls the user mode thread manager routine specific to the upcall type. The thread manager may save the context of the current user mode thread and switch a different one into the kernel thread.
5. Control does not return to EXE$CALLBACK_JACKET until the thread manager switches the preempted user mode thread back into execution. When control does return, EXE$CALLBACK_JACKET must restore the state of PKTA$V_TM_ACTIVE at entry. If it was clear, EXE$CALLBACK_JACKET temporarily disables user mode AST delivery because the thread most likely no longer has an active user mode AST blocking

AST delivery. It then clears the bit and restores the previous state of user mode AST delivery.

### 5.9.2 Thread Quantum End

OpenVMS scheduling enforces a quantum on each kernel thread, partly to implement round-robin scheduling and partly to ensure minimum memory residence after a process has been inswapped. CPU time, as well as a fixed time for each voluntary wait the kernel thread incurs, is charged against the quantum. Chapter 2 provides further details.

Similarly, OpenVMS multithreading support provides for user mode thread quantum. When a user mode thread manager registers itself, it optionally specifies a value for user mode thread quantum. DECthreads specifies a value of two 10-millisecond ticks. Only CPU time, not voluntary wait time, is charged against user mode thread quantum, however. User mode thread quantum helps the thread manager implement round-robin scheduling by triggering a user mode thread reschedule attempt at regular intervals.

If the thread manager specifies a user mode thread quantum value, the executive stores it in PCB$L_TQUANTUM and the initial kernel thread's KTB$L_TQUANT. It also allocates an ACB to deliver a quantum-end upcall, storing its address in KTB$L_TQUANT_ACB. Whenever a new kernel thread is created in this process, an ACB is allocated for its thread quantum-end upcalls and its KTB$L_TQUANT is initialized.

The interval timer interrupt service routine (see Chapter *Time Support*) charges a 10-millisecond tick that occurs in process context against the current kernel thread's quantum. If PCB$L_TQUANTUM is nonzero, the routine also charges the tick against KTB$L_TQUANT. If the charge results in kernel mode or user mode thread quantum end, the interval timer interrupt service routine requests a software timer interrupt.

The software timer interrupt service routine deals with user mode thread quantum end by calling SCH$THREADQ_END, in module TIMESCHDL, which takes the following steps:

1. It resets KTB$L_TQUANT.
2. It calls EXE$CHECK_UPCALL to see whether an upcall is possible.
3. If an upcall is possible and if the thread quantum end ACB is not in use, SCH$THREADQ_END increments KTB$L_TQUANT_ACB as a flag that the ACB is in use. It stores the kernel thread's KTB$L_PID in ACB$L_PID and queues the ACB.
4. SCH$THREADQ_END returns to the software timer interrupt service routine.

The thread quantum end ACB describes a user mode AST with a piggyback special kernel mode AST. When the AST is delivered, the special kernel mode AST routine, SCH$THREADQ_KAST in module RSE, is entered first.

Its only function is to clear the low bit of KTB$L_TQUANT_ACB, the in-use flag for the ACB.

The user mode AST procedure EXE$QUANTUM_AST, in module TM_SUPPORT, calls EXE$CALLBACK_JACKET, passing it the upcall type of thread quantum end. As described in Section 5.9.1, EXE$CALLBACK_JACKET calls the thread manager routine specific to thread quantum end.

If appropriate, the DECthreads thread manager saves the current user mode thread's context and switches to another thread. Control cannot return to EXE$CALLBACK_JACKET until DECthreads reschedules the user mode thread at its next turn to run.

The executive provides the $FORCE_QUANTUM_END system service to enable a user mode thread manager to force thread quantum end. The system service procedure, EXE$FORCE_QUANTUM_END in module TM_SUPPORT, confirms that the target kernel thread is part of the same process, gets the address of its KTB, and checks whether a user mode thread quantum was specified for this process. If so, it clears KTB$L_TQUANT. Otherwise, it clears KTB$L_QUANT. The next time an interval timer interrupt occurs while this kernel thread is running, it will experience quantum end.

The DECthreads thread manager uses this service to force a context switch from a running user mode thread to the null thread (see Section 5.10.1) during debugging. This enables the debugger to access the saved register state of the formerly running thread.

### 5.9.3 Event Flag Setting Events

Local event flags and flag EFN$C_ENF (see Chapter 3) are processwide resources. When a user mode thread manager registers itself and specifies upcall routines for event flag waits and settings, the executive allocates an ACB for upcalls (see Section 5.4.2). The executive notifies a user mode thread manager registered in a process whenever a user mode thread in that process requests a wait for local flags that are not set. The thread manager may save the context of that user mode thread and context-switch another one into the kernel thread. Consequently, when any of these flags are set, the executive must also notify the user mode thread manager that the flags have been set so that the thread manager can unblock user mode threads waiting for local flags that have now been set.

Moreover, inner mode code can request $SYNCH and event flag wait services. A kernel thread requesting a wait for these flags from an inner mode is placed into a local event flag (LEF) wait. Consequently, when any of these flags is set, the executive must also check every kernel thread that is in an LEF wait in a multithreaded process to see if its wait has now been satisfied.

SCH$POSTEF, in module POSTEF, the routine that sets all event flags, is responsible for notifying the user mode thread manager through an AST upcall and for reporting event flag setting events for kernel threads that should be unwaited. SCH$POSTEF is called with the IPID of the kernel thread for

which an event flag is being set, the kernel thread that requested the operation whose completion is being signaled by the setting of this flag.

SCH$POSTEF calls EXE$CHECK_UPCALL to determine if a user mode thread manager has been registered in the process (see Section 5.4.2) with an event flag setting upcall. If one has, SCH$POSTEF sets a bit corresponding to the event flag in the process's PCB$Q_POSTEF field. If flag EFN$C_ENF is being set, SCH$POSTEF sets PCB$V_EVENT_NO_FLAG in PCB$L_THREAD_EVENTS.

SCH$POSTEF tests and sets the low bit of PCB$L_POSTEF_ACB, which is the in-use bit for the event flag upcall ACB. If the ACB was not in use, SCH$POSTEF stores the address of the kernel thread's PKTA$Q_POSTEF in the ACB as the AST argument and queues the ACB to the kernel thread for which the event flag is being set. From the time the AST is queued until it is delivered, any additional local flags set by SCH$POSTEF will be reflected as additional bits set in PCB$Q_POSTEF.

If the process has more than one kernel thread, SCH$POSTEF also checks the scheduling state of every kernel thread in the process to see whether a possible event flag wait has been satisfied by the setting of this flag. Chapter 3 describes this in further detail.

The event flag setting AST has an associated piggyback special kernel mode AST routine, EXE$POSTEF_AST in module POSTEF. Running in the context of the kernel thread for which the event flag was being set, it clears the low bit of PCB$L_POSTEF_ACB to signal that the ACB is no longer in use. It executes a memory barrier instruction to order clearing the bit with the subsequent reads. It executes a load-locked instruction to read PCB$Q_POSTEF and then clears it with a store-conditional instruction to ensure no concurrent attempt by SCH$POSTEF to set more bits. If other flags were set while the ACB was queued, the thread manager will be notified of all of them. It gets the value of PCB$L_THREAD_EVENTS in a similar manner.

EXE$POSTEF_AST copies the contents it read from the two PCB fields to PKTA$Q_POSTEF and PKTA$Q_THREAD_EVENTS. Note that the per kernel thread area is in process-private address space and cannot be directly accessed by SCH$POSTEF.

The user mode AST procedure EXE$SETEF_AST, in module TM_SUPPORT, calls EXE$CALLBACK_JACKET, passing it the upcall type of event flag setting.

DECthreads checks whether any of the set event flags can allow unblocking of a user mode thread waiting for flags. If so, it unblocks each such user mode thread.

## 5.10 DECTHREADS THREAD MANAGER OVERVIEW

DECthreads provides several application programming interfaces to support the creation and control of multiple user mode threads of execution in a single address space. DECthreads services comply with the Open Software

Foundation's Distributed Computing Environment standards and conform to the IEEE POSIX approved threads standard 1003.1c-1995 interface.

A user program calls routines documented in the *Guide to DECthreads* to create and synchronize user mode threads. The routine name references in the program are resolved in one or more DECthreads shareable images, in particular, the DECthreads run-time library PTHREAD$RTL.EXE, which supports the POSIX standard interface and includes code that interacts with the OpenVMS executive.

DECthreads is procedure-based code that runs in the context of a single process and coordinates user mode threads within the kernel threads of that process. On a system with multiple processes running user mode multithreading applications, a DECthreads thread manager runs in each such process independently of the other processes. Because the thread manager is implemented as a shareable image, its code pages can be shared by multiple processes, but each process has its own copy of writable data.

The sections that follow briefly describe DECthreads operations on OpenVMS Alpha Version 7.0 systems.

### 5.10.1 DECthreads Initialization

PTHREAD$RTL.EXE is a shareable image with little dependence on other shareable images. Its initialization routine is therefore called prior to those of most other shareable images with which the main image may have been linked. Its initialization routine requests the $INIT_MULTITHREAD and $REGISTER_THREAD_MANAGER system services and initializes DECthreads data. Consequently, other shareable images' initialization routines can create user mode threads as part of their initialization.

When activating a main image linked with PTHREAD$RTL.EXE, the image activator manipulates the addresses in the transfer address array (see Chapter 8). As a result, after all shareable images' initialization routines are called, the DECthreads routine pthread_main is called next, before the main image is entered. This routine creates a DECthreads-specific procedure frame on the initial kernel thread's user mode stack. The initial user mode thread will run on this stack.

The procedure frame is identical to the one DECthreads will create later on each additional user mode thread's stack. Having this call frame on the initial user mode thread's stack enables DECthreads to control the initial user mode thread as it does all other user mode threads, in particular, with respect to handling pthread_exit or cancellation of the initial user mode thread.

The first routine to run in a newly created kernel thread bootstraps DECthreads scheduling for that kernel thread. It creates a user mode null thread and switches to its stack. The null thread runs in the kernel thread when DECthreads has no other runnable user mode thread to schedule in that ker-

nel thread. While the null thread is running, bit PKTA$V_TM_ACTIVE in its kernel thread's PKTA is set.

The null thread first calls the pthread_yield_np routine, which switches to any other higher priority runnable noncurrent user mode thread. If there is no other user mode thread to run in this kernel thread, pthread_yield_np returns. The null thread requests the $HIBER system service. Because the PKTA$V_TM_ACTIVE bit is set, the $HIBER system service does not perform an upcall but instead places the kernel thread running the null thread into hibernation.

Whenever the DECthreads thread manager unblocks a user mode thread or creates a new one and needs an available kernel thread (or "virtual processor"), the thread manager awakens the kernel thread running the null thread. The null thread then repeats its call to pthread_yield_np and is context-switched with the newly runnable user mode thread.

### 5.10.2 User Mode Thread Data Structures

Each user mode thread is described by a DECthreads data structure called a thread control block. It consists of fields that are publicly defined and accessible to applications, and fields that are privately defined and available only to DECthreads run-time library routines. The public part of the structure is called a thread environment block (TEB). To facilitate easy application migration, its layout is the same on all platforms supported by DECthreads. Figure 5.14 shows its layout and illustrates its position relative to the rest of the thread control block.

TEB$PQ_FLINK and TEB$PQ_BLINK link the TEB into queues in the DECthreads scheduler database.

TEB$Q_SEQUENCE identifies the user mode thread. The initial user mode application code is considered the first user mode thread and has a sequence number value of 1. The first user mode thread it creates has a sequence number value of 2; and so on. Debugger commands targeted to a particular thread identify the thread by this sequence number.

TEB$PQ_PER_KT_AREA contains the address of the per kernel thread area associated with this user mode thread. Whenever the user mode thread is context-switched into a different kernel thread, this field is updated.

TEB$PQ_TSD_VALUES contains the address of thread-specific static data associated with this user mode thread. The data is organized as an array of quadwords indexed by key. An application reads and writes a particular data value through the routines pthread_getspecific and pthread_setspecific.

TEB$L_TSD_COUNT contains the number of elements in this thread's array. Each user mode thread initially has ten quadwords. An application can call the routine pthread_key_create to create additional keys. When a thread accesses a value beyond the end of its array but within the number of keys for the process, DECthreads expands the array for that thread.

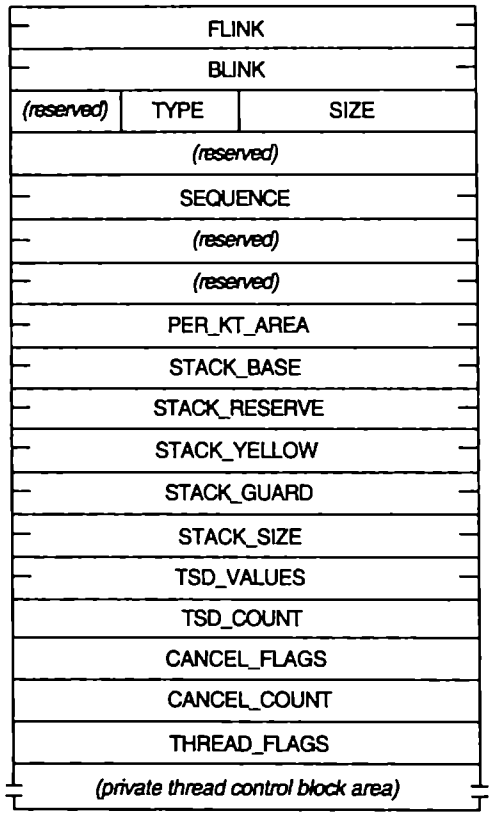



**Figure 5.14**
Layout of a Thread Environment Block (TEB)

TEB$L_CANCEL_FLAGS and TEB$L_CANCEL_COUNT are used in implementing DECthreads cancel operations.

TEB$L_THREAD_FLAGS contains two defined flags:

- TEB$V_ASTACT, when set, specifies that this user mode thread has an active AST. This bit is primarily for maintaining consistent behavior of the run-time library routine LIB$AST_IN_PROG when it is called from a user mode thread. When a user mode thread requests the Get Job/Process Information ($GETJPI) system service specifying item code JPI$_ASTACT, the state of this bit is returned as the user mode AST active bit.
- TEB$V_ASTEN, when set, means that user mode AST delivery is enabled to this user mode thread. This bit is set or cleared by DECthreads in response to a $SETAST upcall. When a user mode thread requests $GETJPI specifying item code JPI$_ASTEN, the state of this bit is returned as the user mode AST enable.

TEB$PQ_STACK_BASE, TEB$PQ_STACK_RESERVE, TEB$PQ_STACK_YELLOW, and TEB$PQ_STACK_GUARD delimit areas of the user mode stack associated with this user mode thread. Figure 5.15 illustrates the various areas of the user mode stack. TEB$Q_STACK_SIZE is the size in bytes

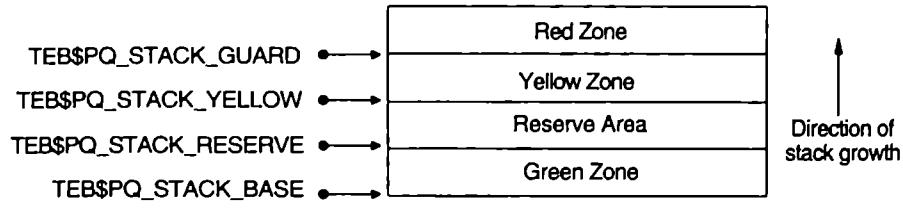



**Figure 5.15**
User Mode Stack Areas

of the usable portion of the stack, that is, the green zone and reserve area in Figure 5.15.

When creating a user mode thread, DECthreads creates a user mode stack for it based on sizes specified by the application. In general, each user mode thread requires its own stack and must run on the same stack throughout its lifetime; a user mode thread therefore cannot use the user stack associated with a particular kernel thread. The initial user mode thread, however, does run on the user mode stack associated with the initial kernel thread because it is an expandable stack. Running the initial user mode thread on an expandable stack increases compatibility for an application to which user mode multithreading is being added.

The size of the green zone is specified by the application through the routine pthread_attr_setstacksize. This part of the stack is used by code running in the user mode thread. The reserve area is required to ensure stack space for a context switch or upcall. The yellow zone currently has a length of 0 bytes and is unused. The default size of the red zone is one page. The application can override the default by calling the routine pthread_attr_setguardsize_np. If the red zone exists, its protection prohibits any access so that it can serve as a guard page.

### 5.10.3 User Mode Thread Identification

Each DECthreads user mode thread can be identified by two values. One, returned by the pthread_create routine in the thread argument, is the address of the TEB. When a user mode thread is current, this value corresponds to its hardware process unique value, the value read by the READ_UNQ PALcode routine.

The other value is the sequence number assigned when the thread is created, the value stored in TEB$Q_SEQUENCE. The debugger uses the sequence number to identify a thread in commands and in display output.

The DECthreads routine pthread_getsequence_np returns the current user mode thread's sequence number.

Note that the IPID and EPID associated with a user mode thread vary in a process with multiple kernel threads. Each time a user mode thread is placed

into execution, it may be running in a different kernel thread and thus have a different IPID and EPID.

### 5.10.4 Thread Manager Scheduling

The DECthreads thread manager runs in user mode. It is not associated with any particular kernel thread and, in fact, may be entered concurrently in multiple kernel threads in a multithreaded process.

It maintains information about all the user mode threads in a process in its own scheduling database. It determines and keeps track of the priorities of user mode threads; these priorities are independent of the kernel thread priorities maintained by the OpenVMS executive and are transparent to the executive. It synchronizes its own accesses from multiple kernel threads to the scheduling database with a DECthreads spinlock. It uses DECthreads mutexes to synchronize its access to other data it maintains on behalf of the process.

There is no communication between OpenVMS scheduling code and the DECthreads thread manager about capability requirements and explicit CPU affinities (see Chapter 2). Code running in a user mode thread can request either the Modify Process User Capabilities ($PROCESS_CAPABILITIES) or the Modify Process Affinity ($PROCESS_AFFINITY) system service to constrain its execution to a subset of SMP members. The DECthreads thread manager, however, would be unaware of these constraints and be unable to take them into consideration when scheduling user mode threads to run.

The DECthreads user mode thread manager switches user mode threads within a single kernel thread in response to events within the application. If the SYSGEN parameter MULTITHREAD has a value equal to or greater than 1, DECthreads is notified through upcalls when events visible to the OpenVMS executive occur that affect the computability of user mode threads.

The DECthreads thread manager can be entered to reschedule both through upcalls from the OpenVMS executive and through calls from DECthreads routines that block and unblock user mode threads.

The OpenVMS executive tests and sets bit PKTA$V_TM_ACTIVE in the per kernel thread area associated with a kernel thread when it is about to perform an upcall to notify the thread manager about a blocking event. The bit prevents recursive reentry into the thread manager if the thread manager itself, already running on behalf of a user mode thread being blocked, incurs a blocking event, for example, a page fault. In such circumstances, the OpenVMS executive places the kernel thread into a wait.

In response to a user mode thread initiated block, DECthreads itself sets PKTA$V_TM_ACTIVE in the per kernel thread area associated with the kernel thread.

Blocking upcalls and user mode thread initiated blocks are synchronous events from the viewpoint of the user mode thread being blocked and the thread manager. In response to either, DECthreads acquires its scheduling spinlock and selects the highest priority runnable user mode thread and saves the context of the blocked user mode thread. For example, it saves PKTA$Q_IEEE_FP_CONTROL in a reserved area of the TEB. It saves integer registers and floating-point registers on the current user mode stack, creating a canonical frame on the user mode stack. It saves the stack pointer in the thread control block. It saves the state of the user mode bit from the AST enable processor register in the thread control block.

Every noncurrent kernel thread has a canonical kernel stack frame (see Chapter 2) so that a single executive routine can restore a kernel thread's registers and place it into execution whether the kernel thread was preempted by a higher priority kernel thread or placed into a wait. Similarly, every noncurrent user mode thread has a canonical user mode stack frame to allow one DECthreads routine to restore all the registers and place the user mode thread into execution.

The thread manager executes the WRITE_UNQ PALcode routine to update the hardware process unique value with the address of the new thread's TEB. It updates the floating-point control register to reflect the new thread's floating-point usage. It updates the user mode AST enable bit based on the new thread's state. It copies information from the new thread's TEB to the user-writable PKTA data. It restores the new thread's user mode stack pointer and restores other registers from its stack. Running in the context of the new user mode thread, it clears PKTA$V_TM_ACTIVE, releases the DECthreads scheduling spinlock, and returns control to the next instruction in that thread.

The thread manager can accept unblocking upcalls at any time. From the viewpoint of the current user mode thread, an unblocking upcall for another thread is generally an asynchronous event delivered as a user mode AST. In response to such an upcall, the thread manager upcall routine records the information about the event and puts the information in a "to do" list. If the DECthreads scheduling spinlock is currently free, the upcall routine acquires the spinlock. It saves the context of the current thread and switches to that of a high-priority internal DECthreads housekeeping thread called the manager thread.

The manager thread requests the $CLRAST system service so that delivery of subsequent upcall ASTs is possible. It then processes each element in the "to do" list, updating the DECthreads scheduling database to reflect all the unblocking events in the list. It then context-switches to the current highest priority user mode thread that is not running.

If the SYSGEN parameter MULTITHREAD has a value of 0, the OpenVMS executive does not perform upcalls. To ensure that the thread manager

receives periodic control in those circumstances, it requests the Set Timer ($SETIMR) system service to deliver an AST after a certain amount of time has elapsed. The timer AST procedure calls DECthreads scheduling code and rerequests the $SETIMR system service. DECthreads cannot detect that the kernel thread running the interrupted user mode thread was in a wait state and may schedule the user mode thread to run again before the wait is satisfied. In that case, the kernel thread is placed back into a wait.

### 5.10.5 Example of DECthreads User Mode Thread Scheduling

Figure 5.16 illustrates a possible DECthreads scheduling sequence in a process with only one kernel thread that runs on a system with MULTITHREAD set to 1 or higher. The wavy lines represent the execution of arbitrary application code within each user mode thread. Scheduling events are represented by rectangles. Whenever an event occurs that might cause the kernel thread to be placed into a wait in user mode, the executive performs an upcall to DECthreads, which selects another user mode thread to run.

Whenever an event occurs that might cause a user mode thread to be unblocked, the executive performs an upcall to DECthreads. In this particular example, the user mode thread being unblocked is always of higher priority than the thread currently running. Consequently, DECthreads context-switches the current thread for the one being unblocked.

On a system with MULTITHREAD set to 0, any event that blocks a user mode thread (for example, a page fault), causes the kernel thread on which the user mode thread is running to be placed into a wait. An unblocking event changes the state of the waiting kernel thread to computable. Timer-based DECthreads code could also cause rescheduling of user mode threads.

### 5.10.6 User Mode ASTs

Sections 5.8.2.2 and 5.8.2.3 describe the queuing of user mode ASTs and the upcall made from the user mode AST delivery interrupt service routine.

A user mode AST delivery upcall is done by delivering a user mode AST to the DECthreads thread manager. The parameter associated with the AST is the address of PKTA$Q_AST_ADDRESS, which contains the AST procedure address. PKTA$Q_AST_PARAMETER contains the argument to be passed to that AST procedure. PKTA$Q_AST_THREAD_ID, if nonzero, identifies the user mode thread for which the AST is intended. In OpenVMS Alpha Version 7.0, the thread ID is always zero. As previously described, the upcall routine records this information and puts it on the "to do" list.

When the manager thread processes this "to do" list item, it checks whether the thread ID is zero (untagged) or whether it identifies a particu-

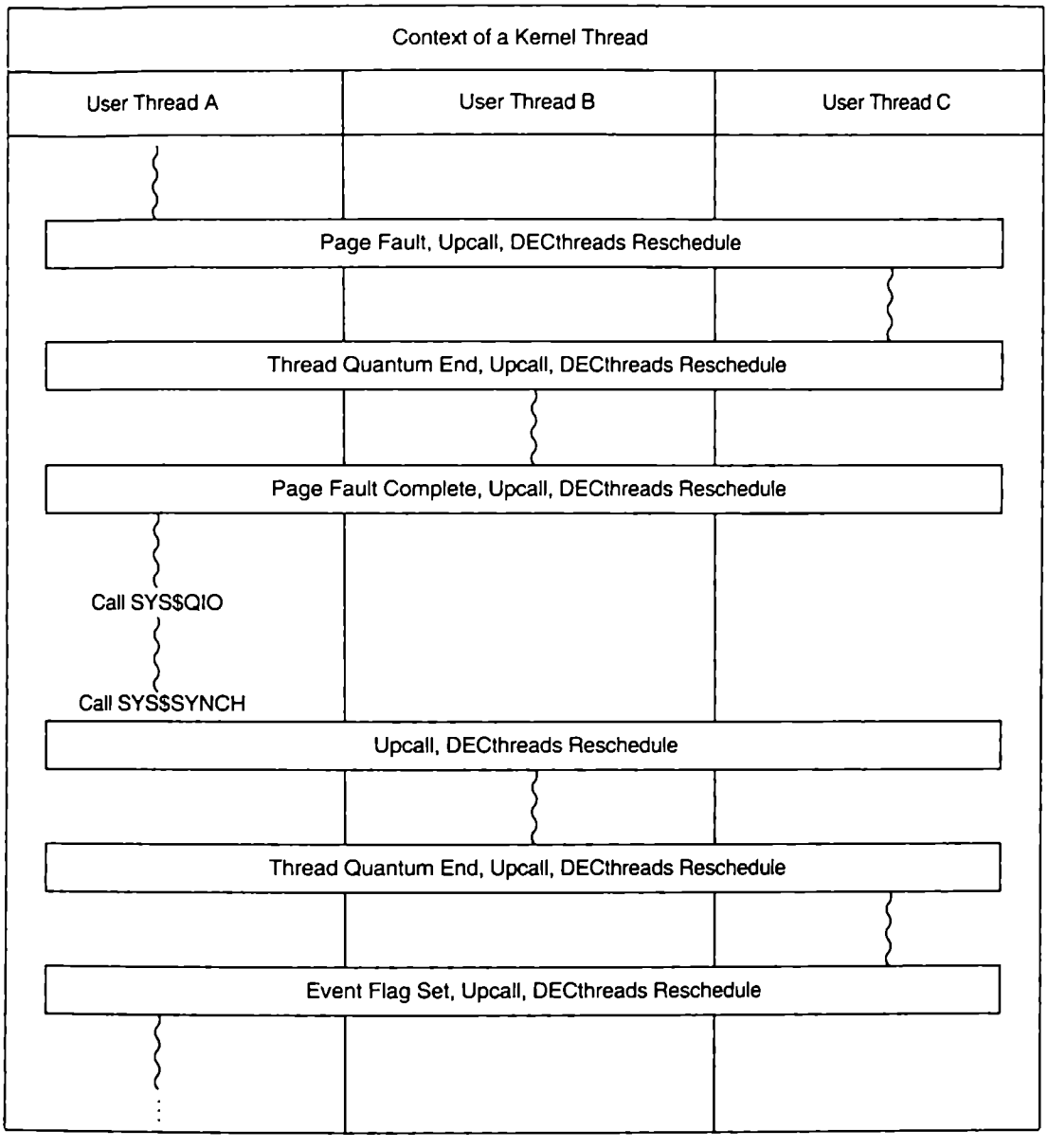

Context of a Kernel Thread
User Thread A
User Thread B
User Thread C
Page Fault, Upcall, DECthreads Reschedule
Thread Quantum End, Upcall, DECthreads Reschedule
Page Fault Complete, Upcall, DECthreads Reschedule
Call SYS$QIO
Call SYS$SYNCH
Upcall, DECthreads Reschedule
Thread Quantum End, Upcall, DECthreads Reschedule
Event Flag Set, Upcall, DECthreads Reschedule

**Figure 5.16**
Example User Mode Thread Flows

lar user mode thread. If the AST was untagged, the manager thread checks that no user mode thread has disabled AST delivery by requesting $SETAST. If any has disabled AST delivery, the AST cannot be delivered until each thread that has disabled AST delivery reenables it.

Note that if no user mode thread has disabled AST delivery, user mode threads and one or possibly more user mode AST procedures can run in parallel.

The manager thread delivers each untagged AST to the initial user mode thread. The initial user mode thread is scheduled preferentially during AST delivery to ensure timely AST delivery. Delivery of tagged ASTs is done at the current priority of the user mode thread for which the AST is intended.

Delivering an AST to a user mode thread basically means that when the thread is next placed into execution, it executes DECthreads code that sets TEB$V_ASTACT and then calls the AST procedure. When the AST procedure returns, DECthreads code clears TEB$V_ASTACT and reenters the code that was running when the user mode thread was last taken out of execution. If the user mode thread was blocked prior to the AST delivery and the blocking condition has not changed, the user mode thread will block again.

DECthreads must also be notified when a user mode thread requests $SETAST to disable or enable user mode ASTs. Basically, because much communication between the executive and DECthreads occurs through user mode ASTs, if the executive disabled user mode AST delivery in a kernel thread, all upcalls in that kernel thread made through ASTs would be blocked until AST delivery were reenabled.

Instead, the executive notifies DECthreads that the current user mode thread has requested $SETAST. In response, DECthreads checks that no user mode thread is executing an AST and clears TEB$V_ASTEN in that user mode thread's TEB. If any user mode threads are executing AST procedures, the user mode thread trying to disable delivery is stalled until the AST procedures complete.

Similarly the executive notifies DECthreads of a reenabling so that DECthreads can set TEB$V_ASTEN.

### 5.10.7 User Mode Thread Exit

In response to an exit upcall, DECthreads first unblocks its user mode thread reserved for running exit handlers. Because this thread has high priority, it is likely to be scheduled very soon.

DECthreads then calls the routine pthread_exit, which enables nested routines on the stack of the thread requesting $EXIT to release any DECthreads or application resources held by this user mode thread. Releasing these now prevents possible deadlocks later when exit handlers are called. In the process of informing nested routines of the impending exit, pthread_exit unwinds the thread's stack. Control returns to DECthreads, which switches to another user mode thread.

The user mode exit handler thread sets PKTA$V_TM_ACTIVE and then itself requests the $EXIT system service. Because PKTA$V_TM_ACTIVE is set, the $EXIT system service does not attempt another upcall and instead clears PKTA$V_TM_ACTIVE, as described in Section 5.9.1. The $EXIT system service then requests $EXIT_INT, which begins by calling user mode exit handlers (see Chapter 8).

## 5.11 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIB]FREDDEF.SDL
[LIB]IMSEMDEF.SDL
[LIB]PCBDEF.SDL
[LIB]PKTADEF.SDL
[LIB]TMCDDEF.SDL
[STARLET]TEBDEF.SDL
[SYS]ASTDEL.MAR
[SYS]ASTDEL_STACK.M64
[SYS]SYSTEM_SERVICE_DISPATCHER.M64
[SYS]SYSTEM_SERVICE_IMSEM.MAR
[SYS]SYSWAIT.MAR
[SYS]TM_SUPPORT.BLI

# 6 Process Control and Communication

I was alone and unable to communicate with anyone. I did not know the names of anything. I did not even know things had names. Then one day, after she had tried a number of approaches, my teacher held my hand under the water pump on our farm. As the cool water ran over my hand and arm, she spelled the word water in my other hand. She spelled it over and over, and suddenly I knew there was a name for things and that I would never be completely alone again.

Helen Keller, *The Story of My Life*

The executive provides a number of services that allow one process to control the execution of another. It also provides a variety of mechanisms by which processes can obtain information about each other and communicate with one another.

Process control system services enable a process to affect its own scheduling state or that of another process, either on the local system or on other VMScluster nodes. These services also enable a process to alter some of its own characteristics (such as name or priority). The process information system services allow a process to obtain detailed information about other processes, both on the local system and on other VMScluster nodes. This chapter describes the implementation of the process control and process information system services.

Kernel threads, introduced in OpenVMS Alpha Version 7.0, provide multiple execution contexts for a process. Some process control and information services affect the process as a whole, some affect an individual kernel thread, and some can affect either, depending on the user-specified arguments. This chapter describes the role of kernel threads in the process control and process information system services; Chapter 5 describes kernel threads in detail.

Communication mechanisms available to processes include event flags, mailboxes, lock management system services, global shared data sections, and shared files. Other chapters describe the implementation of these mechanisms. This chapter briefly discusses the manner in which a process might use these mechanisms to communicate with another process.

Table 6.1 summarizes the system services related to process control and process information.

Table 6.1 Summary of Process System Services

| *Service Name* | *Scope of Processes Affected* | *Privileges Checked* |
|---|---|---|
| Hibernate ($HIBER) | Issuing kernel thread [1] | None |
| Wake Process from Hibernation ($WAKE) | Process in same VMScluster | GROUP or WORLD |
| Schedule Wakeup ($SCHDWK) | Process in same VMScluster | GROUP or WORLD |
| Cancel Wakeup ($CANWAK) | Process in same VMScluster | GROUP or WORLD |
| Suspend Process ($SUSPND) | Process in same VMScluster | GROUP or WORLD |
| Resume Process ($RESUME) | Process in same VMScluster | GROUP or WORLD |
| Exit ($EXIT) | Issuing process | None |
| Force Exit ($FORCEX) | Process in same VMScluster | GROUP or WORLD |
| Create Process ($CREPRC) | Process in same VMScluster | DETACH for different user identification codes |
| Delete Process ($DELPRC) | Process in same VMScluster | GROUP or WORLD |
| Set AST Enable ($SETAST) | Issuing kernel thread | Access mode check |
| Set Power Recovery AST ($SETPRA) | Issuing process | Access mode check |
| Set Priority ($SETPRI) | Process in same VMScluster | ALTPRI and either GROUP or WORLD |
| Set Process Name ($SETPRN) | Issuing process | None |
| Set Resource Wait Mode ($SETRWM) | Issuing process [2] | None |
| Set System Service Failure Exception Mode ($SETSFM) | Issuing process [2] | Access mode check |
| Set Process Swap Mode ($SETSWM) | Issuing process [2] | PSWAPM |
| Reschedule Process ($RESCHED) | Issuing kernel thread | None |
| Get Job/Process Information ($GETJPI) | Process in same VMScluster | GROUP or WORLD |
| Process Scan ($PROCESS_SCAN) | Process in same VMScluster | GROUP or WORLD |
| Modify Process Implicit Affinity ($SET_IMPLICIT_AFFINITY) | Process on same node | ALTPRI and either GROUP or WORLD |
| Modify Process User Capabilities ($PROCESS_CAPABILITIES) | Process on same node | ALTPRI and either GROUP or WORLD |
| Modify Process Affinity ($PROCESS_AFFINITY) | Process on same node | ALTPRI and either GROUP or WORLD |

[1] As part of the $CREPRC system service, a process can specify that the process being created hibernate before a specified image executes.

[2] Through the $CREPRC system service, a process can be created with this characteristic.

## 6.1 REQUIREMENTS FOR AFFECTING ANOTHER PROCESS

Before a process can obtain information on another process or alter it in any way, it must have a means of uniquely identifying the process within a VMScluster system. In addition, it must have appropriate privileges or user identification code (UIC) based access to the process.

Process identification and privilege checking are centralized in the routine EXE$NAM_TO_PCB, in module SYSPCNTRL. Process control and process

information system services that can affect processes other than their requestor all call EXE$NAM_TO_PCB; thus they all identify processes and check privileges in the same manner.

In early versions of OpenVMS, the scope of the process control and process information system services was restricted to the local node; the scope became the entire VMScluster with the addition of clusterwide process services (CWPS) routines. These routines provide a transparent mechanism by which a process can affect a target process on another VMScluster node. Section 6.1.4 contains more information on CWPS routines.

### 6.1.1 Identifying the Target Process

Process control and process information system services have arguments that specify the target process by process name or process ID (PID). If the process requesting the system service does not specify the target process using one or the other of these arguments, the target process is assumed to be the requesting process.

Process name is always implicitly qualified by UIC group. That is, a process can identify by name only processes within the same UIC group as itself. The PRCNAM argument can identify a process on another VMScluster node. It can include up to six characters for the node name, followed by a double colon.

Two forms of PID identify a process: an internal PID (IPID) and an extended PID (EPID). The IPID, stored in PCB$L_PID or KTB$L_PID, uniquely identifies a process and a kernel thread on a single node. For all kernel threads in a process, the low word of the IPID contains the index of the same process control block (PCB) in the local PCB vector. The high word of the IPID, the sequence number, identifies the kernel thread. The EPID uniquely identifies a process and a kernel thread in a VMScluster system by including a VMScluster node identifier. The EPID is stored in PCB$L_EPID or KTB$L_EPID. Each of a process's kernel threads includes the same VMScluster node identifier and the index of the same PCB in its EPID. Chapter 7 describes the layout and creation of the IPID and EPID.

Because the IPID is only relevant to kernel mode code on the local node, most system utilities, such as SHOW SYSTEM and MONITOR, display EPIDs. An EPID is passed as a system service argument to identify a process by its PID.

A legitimate EPID never has its high-order bit set; the Get Job/Process Information ($GETJPI) and Process Scan ($PROCESS_SCAN) system services can thus use a negative value in an EPID field as a wildcard indicator.

### 6.1.2 Processes and Kernel Threads

Using SYSGEN parameters, an OpenVMS system can be configured to execute processes as

- Single-threaded

- Multithreaded with only one kernel thread per process
- Multithreaded with multiple kernel threads per process

Each process is described by a PCB, which contains information such as the process name, the access rights block (ARB) and UIC, the lock queue, and the kernel thread count. Each kernel thread is described by a kernel thread block (KTB), which contains information about the thread's scheduling state and priority, its CPU affinity, and the internal and extended forms of thread identification. Regardless of how the system is configured, each process has at least one kernel thread, called the initial kernel thread. The KTB for this initial kernel thread overlays the PCB. A field within the PCB can be referenced either by its symbolic PCB name (e.g., PCB$L_STS) or by its KTB name (e.g., KTB$L_STS).

All kernel threads in a process share the same process name, which is stored in the PCB rather than in the KTB. Therefore, process name is insufficient to identify an individual kernel thread. Each kernel thread, however, has a unique IPID and EPID.

A kernel thread needs no privileges to affect any other kernel thread sharing its process context or to affect the process as a whole.

Chapter 5 contains more detail on kernel threads.

### 6.1.3 Locating the Process and Checking Privileges

Regardless of how the target of a process control or process information service is specified, the executive must determine whether the process exists within the VMScluster system and whether the requesting process has the ability to affect the target. These two checks are centralized in EXE$NAM_TO_PCB.

Some of the system services that use EXE$NAM_TO_PCB affect the process as a whole, some affect an individual kernel thread, and some affect either, depending on the user-specified arguments.

EXE$NAM_TO_PCB's argument list includes the EPID of the target kernel thread and the process name from the process control system service's PRCNAM argument.

- When neither argument is specified, the most common case, the requesting kernel thread is also the target. EXE$NAM_TO_PCB is optimized for this case and returns both the PCB and the KTB of the requesting kernel thread.
- When both arguments are specified, EXE$NAM_TO_PCB ignores the process name and uses the EPID to identify both the target process PCB and the target kernel thread KTB.
- When only the process name is present, EXE$NAM_TO_PCB returns the target PCB and the KTB of the process's initial kernel thread. Since the

process's kernel threads share the same process name, the process name is insufficient to identify an individual kernel thread.

EXE$NAM_TO_PCB performs the following:

1. If the requesting kernel thread is also the target, privilege checks are unnecessary; EXE$NAM_TO_PCB returns successfully to the system service with the IPID, the PCB address, the KTB address, and optionally the EPID, holding the SCHED spinlock.
2. If the EPID is specified, EXE$NAM_TO_PCB attempts to locate the target kernel thread using the EPID. If the EPID is not specified, EXE$NAM_TO_PCB attempts to locate the target process using the process name.

   If the EPID or process name indicates a valid local process, EXE$NAM_TO_PCB acquires the SCHED spinlock, raising interrupt priority level (IPL) to IPL$_SCHED. It locates the PCB and the KTB and proceeds to step 3.

   If the target process is not valid locally, the EPID or the process name must designate a legitimate remote VMScluster node. Only the remote node can determine if the target process is valid.

   - If the EPID or process name indicates that the target process is on a valid VMScluster node, EXE$NAM_TO_PCB returns the error status SS$_REMOTE_PROC. Section 6.1.4 describes the steps taken to locate a target process on a remote node.
   - If the EPID specifies an unknown VMScluster node, EXE$NAM_TO_PCB returns the error status SS$_NONEXPR, which becomes the system service's return status.
   - If the process name specifies an unknown VMScluster node, EXE$NAM_TO_PCB returns the error status SS$_NOSUCHNODE.
   - If the node name portion of the process name is in an incorrect format, EXE$NAM_TO_PCB returns the error status SS$_IVLOGNAM.
3. For a local target process, EXE$NAM_TO_PCB calls EXE$CHECKS_PROCESS_ACCESS, in module SYSPCNTRL, to determine whether the requesting process has the ability to examine or modify its target and to perform a process control security audit if necessary.

   EXE$CHECKS_PROCESS_ACCESS returns a success status if any of the following are true:

   - The requesting and target processes are in the same job tree, that is, they share a job information block (JIB).
   - The requesting and target processes have the same UIC.
   - The requesting process has WORLD privilege.
   - The requesting and target processes are members of the same UIC group and the requesting process has GROUP privilege.

4. If EXE$CHECKS_PROCESS_ACCESS returns a success status, EXE$NAM_TO_PCB returns successfully to the system service, still holding the SCHED spinlock.

   Otherwise, EXE$NAM_TO_PCB releases the SCHED spinlock and returns the error status SS$_NOPRIV, which becomes the system service's return status.

### 6.1.4 Servicing a Request for a Remote Kernel Thread

EPID or PRCNAM argument validation for a remote kernel thread occurs in two steps. The VMScluster node identifier part is validated first, followed by the PCB vector slot and the KTB vector slot on the identified VMScluster node. Note that all kernel threads in a process execute on the same VMScluster node.

A process control or process information system service calls EXE$NAM_TO_PCB to locate its target kernel thread. If EXE$NAM_TO_PCB does not locate the target kernel thread on the local node, it calls a CWPS routine to verify that the node identified by the EPID or PRCNAM argument exists within the VMScluster system. Executive code running on the identified node must subsequently confirm the existence of the target kernel thread.

If the CWPS routine successfully identifies the remote VMScluster node, it returns the error status SS$_REMOTE_PROC. In response to this status, the system service procedure routes the request to CWPS. CWPS allocates and initializes a structure to describe the service request and the requesting process, and transmits it to the remote node using system communication services (SCS). If the system service is a synchronous one, such as $SUSPND, the requestor enters the RSN$_CLUSRV resource wait state until a response is received from the remote node. For a system service such as $GETJPI[W], the status SS$_NORMAL is returned to the requestor for the asynchronous form or to a synchronization routine for the synchronous form.

On the remote node, a CWPS dispatch routine executing in system context receives the service request. It allocates a composite structure to describe the request locally. It then queues a kernel mode asynchronous system trap (AST) to the CLUSTER_SERVER process, determining the procedure value of the AST routine from the function to be performed; for example, a process control function causes the CLUSTER_SERVER process to execute CWPS$SRCV_PCNTRL_AST, in module [SYSLOA]CWPS_SERVICE_RECV.

For a typical process control function, the CLUSTER_SERVER process initializes a structure that mimics the PCB of the requesting process and calls EXE$NAM_TO_PCB. Thus, EXE$NAM_TO_PCB performs privilege and access checks, which require a PCB, regardless of whether a request is remote or local. If EXE$NAM_TO_PCB detects an error, the CWPS routine returns the error status to the requesting process on the original node. Otherwise, it

requests the system service on behalf of the requesting process. The system service returns status and information to the CWPS routine, which transmits that information to a CWPS receiver routine on the initiating node. This routine returns the status and data to the original requestor of the system service. (In the case of an asynchronous service like $GETJPI, the standard kernel mode AST is delivered at service completion.)

Figure 6.1 shows this sequence of events, slightly simplified, for an asynchronous process information system service such as $GETJPI.

## 6.2 PROCESS INFORMATION SYSTEM SERVICES

The process information system services, $GETJPI and $PROCESS_SCAN, return selected information about a process or group of processes within a VMScluster system.

The $PROCESS_SCAN system service functions as an adjunct to the $GETJPI system service. It creates and maintains a search context that filters the information returned by $GETJPI. In the traditional form of $GETJPI wildcard processing, an image requests the $GETJPI service from a loop, obtaining information about the next sequential process with each request. The image tests the returned information to decide whether the process really is of interest; for example, an image looking for all processes belonging to a particular user name obtains the user name field through the $GETJPI service and compares each returned user name with its desired user name.

The $PROCESS_SCAN service simplifies this path; an image requests the service to record its search criteria in a context block and passes the context block address on subsequent $GETJPI requests. When the $GETJPI system service procedure is passed a context block address, it calls process scan subroutines for the actual processing. Its requestor only receives information on processes matching the search criteria and is not required to filter the data itself.

With the introduction of kernel threads in OpenVMS Alpha Version 7.0, the $PROCESS_SCAN and $GETJPI services allow a requestor to specify whether the search should return information about a process's initial kernel thread and all of the process's other kernel threads, or whether the search should ignore kernel threads and remain at the process level (the default).

### 6.2.1 Data Structures Related to the $PROCESS_SCAN System Service

The $PROCESS_SCAN system service uses fields in the process header (PHD), context blocks, $GETJPI buffer areas, and CWPS structures to service requests.

An image can request the $PROCESS_SCAN service multiple times with different search criteria to create multiple context blocks. For example, to search a VMScluster system, the image could either create one context block

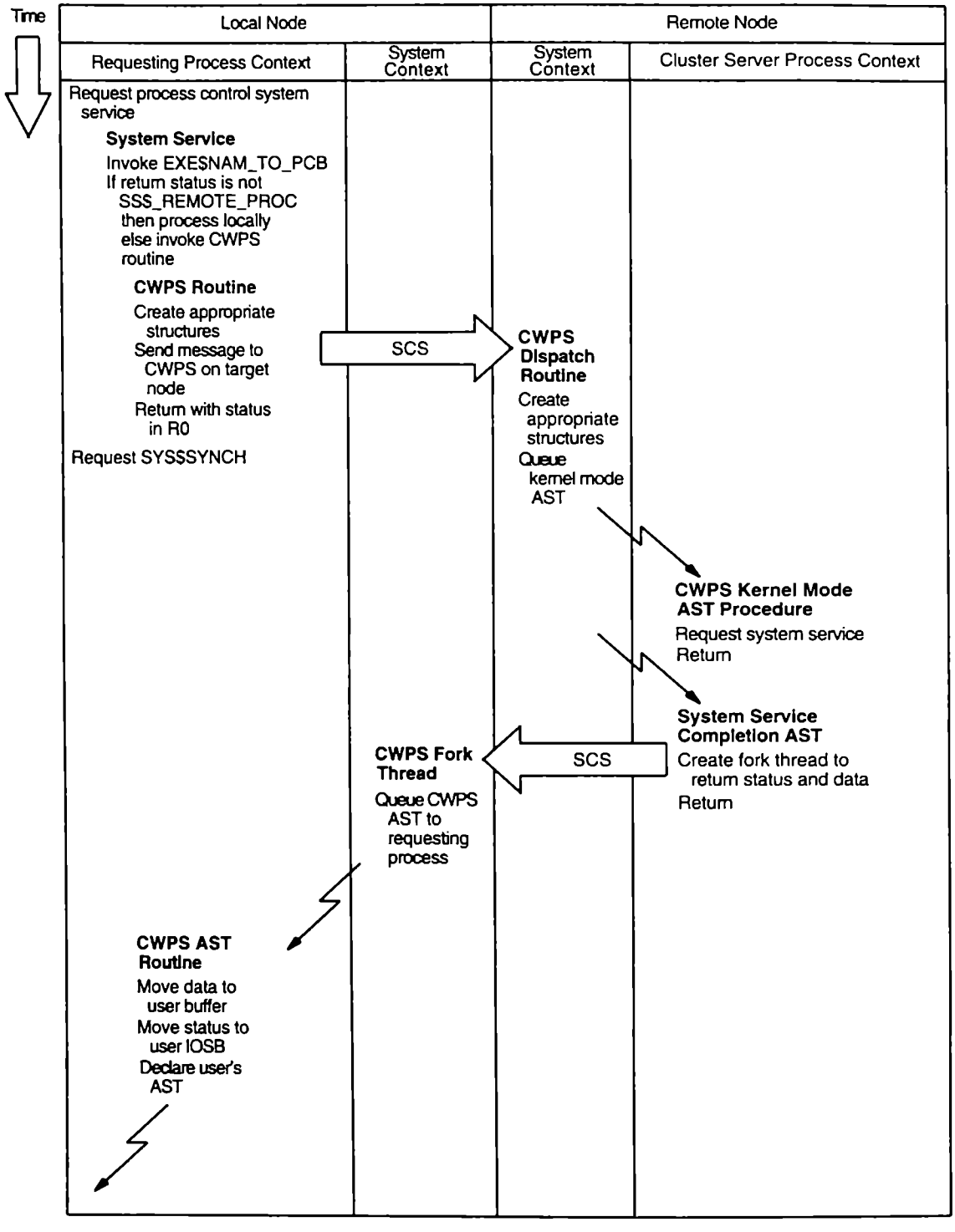



Figure 6.1
CWPS Remote Request Processing

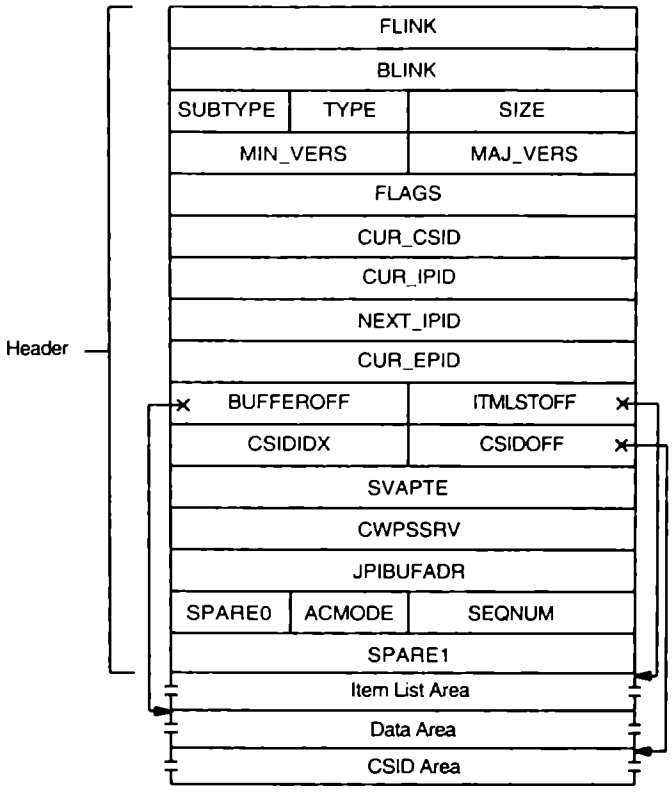



**Figure 6.2**
$PROCESS_SCAN Context Block (PSCANCTX)

matching all cluster nodes or create a separate context block for each node and conduct the remote scans in parallel. The PHD contains a listhead for a process's context blocks at offset PHD$Q_PSCANCTX_QUEUE. At PHD$L_PSCANCTX_SEQNUM, the PHD contains a sequence number that matches the value in the PSCANCTX$W_SEQNUM field of valid context blocks.

The context block, pictured in Figure 6.2, is the primary data structure created and maintained by the $PROCESS_SCAN service. The $PSCANCTXDEF macro defines its header; the size of the structure varies. The item list and data areas follow the header. They contain copies of the $PROCESS_SCAN items comprising the process filter and the associated comparison data. PSCANCTX$W_ITMLSTOFF and PSCANCTX$W_BUFFEROFF contain the byte offsets from the start of the PSCANCTX structure to these areas. The $PROCESS_SCAN service allocates this structure from the process allocation region.

If a search involves VMScluster nodes other than the local node, the context block includes a cluster system ID (CSID) area. This area contains the CSID of each node where the search is to be conducted. PSCANCTX$W_CSIDOFF contains the byte offset from the start of the PSCANCTX structure to this area. PSCANCTX$L_CUR_CSID stores the CSID of the node

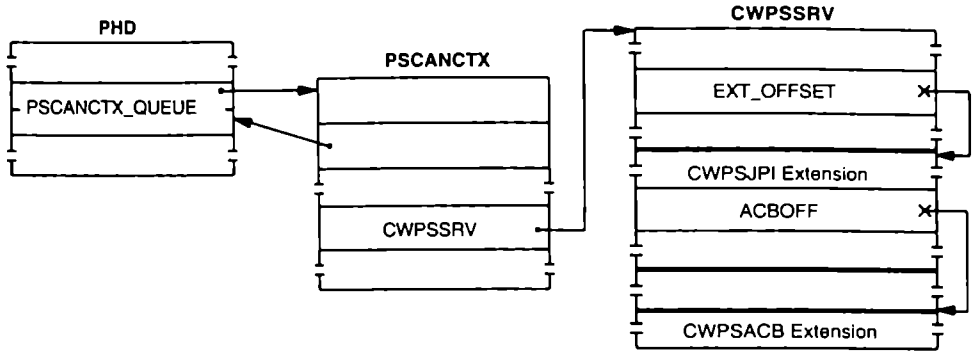



**Figure 6.3**
Structure Linkage

currently being scanned, with zero indicating the local node. The context block fields PSCANCTX$L_CUR_IPID, PSCANCTX$L_CUR_EPID, and PSCANCTX$L_NEXT_IPID track local PCB and KTB vector scans (see Section 6.2.3).

Only one SGETJPI request at a time can use a particular context block to reference other VMScluster nodes. PSCANCTX$V_BUSY in PSCANCTX$L_FLAGS locks the context block; Section 6.2.3 describes its use.

When a remote node is scanned, the offset PSCANCTX$L_CWPSSRV contains the address of a structure whose symbolic offsets are defined by the $CWPSSRV macro. Allocated from nonpaged pool, this variable-sized structure contains information to be passed to the remote node by CWPS. CWPSSRV$L_EXT_OFFSET contains the index to a CWPSSRV extension created for a $GETJPI request, defined by the $CWPSJPI macro. Figure 6.3 shows this linkage.

To execute more efficiently on a wildcard request in a VMScluster system, an image can request that the $PROCESS_SCAN and $GETJPI services bundle information about several target processes rather than return the information one process at a time. If the image requests this $GETJPI buffering, the $PROCESS_SCAN service allocates a buffer from the process allocation region and stores its address in PSCANCTX$L_JPIBUFADR. This variable-sized structure, whose header offsets are defined by the $PSCANBUFDEF macro, contains a copy of the requested $GETJPI item codes followed by the area where returned data is stored.

### 6.2.2 $PROCESS_SCAN System Service

The $PROCESS_SCAN system service procedure, EXE$PROCESS_SCAN in module PROCESS_SCAN, executes in kernel mode. The service includes additional routines in modules PROCESS_SCAN_ITMLST and PROCESS_SCAN_CHECK.

The service has two arguments: PIDCTX and ITMLST.

- PIDCTX is a longword in which the service returns the address of the context block. The system service requestor must pass the returned PIDCTX argument as the PID input argument to the $GETJPI system service.
- ITMLST is the address of an item list composed of one or more entries. Each entry contains the coded value of a selection criterion for $PROCESS_SCAN, either the value or the address of the item, and flags controlling the manner in which the $PROCESS_SCAN and $GETJPI services use the item.

EXE$PROCESS_SCAN takes the following steps:

1. It verifies that the PIDCTX argument was supplied and that it specifies a location writable from the access mode of the requestor. If not, it returns the error status SS$_INVSRQ or SS$_ACCVIO, as appropriate.
2. If EXE$PROCESS_SCAN discovers that the PIDCTX argument contains the address of a previous context block, it removes the context block from the process's queue of active context blocks and deallocates the block and any data structures linked to it. Otherwise, context blocks are deallocated at image exit.
3. If no ITMLST argument is specified, EXE$PROCESS_SCAN merely returns a success status to its caller, accomplishing no useful work. Otherwise, EXE$PROCESS_SCAN checks each item in the item list for the following conditions:
   - The requested item code is recognized.
   - The length of the buffer is appropriate for the item code.
   - The buffer descriptor and the buffer contents are readable.
   - The flags specified for a particular item code are appropriate.

   In addition, the item list must be well formed; for example, the last item cannot specify the flag PSCAN$V_OR.
4. Some item codes apply to processes; others, like PSCAN$_NODE_CSID and PSCAN$_HW_NAME, apply to VMScluster nodes. If the node is part of a VMScluster system and a specified item code indicates a node context, EXE$PROCESS_SCAN calls a CWPS routine to create a list of the current VMScluster nodes and their characteristics. From this, it eliminates nodes that do not meet the search criteria and constructs a table of the remaining CSIDs. Only processes on these nodes are scanned.

   While checking the item list, EXE$PROCESS_SCAN noted whether the user requested kernel-thread-specific items or PSCAN$V_THREAD was set in the PSCAN$_PSCAN_CONTROL_FLAGS item. If so, the other VMScluster nodes meeting the search criteria must either support kernel threads or must not receive this request.
5. EXE$PROCESS_SCAN allocates memory from the process allocation region for the context block. The structure size is the sum of the sizes of the fixed header, the item list entries that apply to processes, their associated data for comparison, and the CSID area (see Figure 6.2). If a

$GETJPI buffer was requested, its size is included in the allocation as well.

6. It increments the process scan sequence number in the PHD, copies that value to the context block, and inserts the new context block into the PHD queue.
7. It initializes the context block, including the offset to the item list (PSCANCTX$W_ITMLSTOFF), the offset to the data area (PSCANCTX$W_BUFFEROFF), the CSID table and the offset to the table (PSCANCTX$W_CSIDIDX), the item list and data areas, the address of the $GETJPI buffer (PSCANCTX$L_JPIBUFADR), the initial process to retrieve (PSCANCTX$L_CUR_IPID), and the flags.

   In particular, it sets the PSCANCTX$V_NEED_THREAD_CAP flag if the user requested kernel-thread-specific items. It sets both the PSCANCTX$V_NEED_THREAD_CAP and PSCANCTX$V_THREAD flags if the PROCESS_SCAN request is to retrieve on a thread-by-thread rather than on a process-by-process basis.
8. Finally, it negates the address of the context block. This indicates a wildcard context yet also locates the context block and differentiates the context from the traditional $GETJPI wildcard indicator, −1 in the high word of the PID. EXE$PROCESS_SCAN returns this value to its requestor in the PIDCTX location.

The $GETJPI service can now be requested to use the context created by $PROCESS_SCAN. The PIDCTX argument is passed to $GETJPI as its PID argument.

### 6.2.3 $GETJPI System Service

The $GETJPI[W] system service provides selected information about a specified process: the process requesting the $GETJPI service (the default), a process explicitly identified by EPID or process name, or the next process located in a wildcard scan. The service can obtain information from the PCB, KTB, JIB, ARB, PHD, and control region.

$GETJPI arguments include the following:

- EFN contains the number of an event flag to be set when the request is complete. If none is specified, event flag 0 is used. EFN$C_ENF is also legal for $GETJPI.
- PID contains the EPID of one process from which to collect information, a traditional $GETJPI wildcard indicator, or a context block from $PROCESS_SCAN.
- PRCNAM contains the node and process name of the target process, used if the process ID is not specified.
- ITMLST contains the address of an item list. The item list can contain multiple entries, each of which includes a code indicating the information

to be returned, the size and address of a buffer to hold the information, and a location to contain the actual size of the returned information. The item list terminates with a longword of zero.

- IOSB contains the address of an I/O status block where $GETJPI records final status information.
- ASTADR and ASTPRM contain the procedure value and parameter of an AST procedure to be called when the request completes.

The $GETJPI system service procedure, EXE$GETJPI in module SYSGETJPI, executes in kernel mode. It performs the following operations:

1. EXE$GETJPI allocates a storage area on the stack for local context items, such as the PCB address and a set of control flags.
2. It obtains the KTB of the current kernel thread.
3. EXE$GETJPI tests the first item list entry, which is the only legal location for the item code JPI$_GETJPI_CONTROL_FLAGS. This item code allows the requestor to limit EXE$GETJPI's behavior with outswapped processes, to restrict AST delivery, and to obtain information on processes that are suspended or marked for deletion. It also allows the requestor to obtain information about a kernel thread rather than a process. By default EXE$GETJPI returns information about the initial kernel thread of the specified process. By specifying the JPI$V_THREAD flag, EXE$GETJPI's searches can include other kernel threads as well.

   EXE$GETJPI copies the control flags, if specified, to its local context area.
4. It checks the PID argument: a value of zero indicates the current process, a positive value indicates an EPID, and a negative value indicates a wildcard specification, either the negated address of a process scan context block or a traditional $GETJPI wildcard.

   - For zero or a positive value, EXE$GETJPI checks whether JPI$V_THREAD was set in JPI$_GETJPI_CONTROL_FLAGS. If so, this request represents a search for all the kernel threads of a single process. EXE$GETJPI sets internal flags indicating the start of a wildcard search for the kernel threads of the specified process, beginning with the initial kernel thread.

     EXE$GETJPI continues processing at step 8.
   - For a negative value, EXE$GETJPI first tests for two special exit conditions, then looks for a context block if necessary.

     If the value is $FFFFFFE0_{16}$, the previous iteration of EXE$GETJPI found the last process applicable to this context. EXE$GETJPI exits, returning the status SS$_NOMOREPROC.

     If the value is $FFFFFFE1_{16}$, the previous iteration of EXE$GETJPI found the last kernel thread applicable to this context. EXE$GETJPI exits, returning the status SS$_NOMORETHREAD.

Otherwise it calls the process scan routine EXE$PSCAN_LOCKCTX, in module PROCESS_SCAN, which negates the argument and compares it to the context blocks in PHD$Q_PSCANCTX_QUEUE.

If no matching context block is found, EXE$GETJPI attempts to process the argument as a traditional $GETJPI wildcard. It obtains the next EPID (see Section 6.2.6) and continues at step 8.

If a matching context block exists, EXE$PSCAN_LOCKCTX attempts to lock the context by setting PSCANCTX$V_BUSY. Only one $GETJPI request at a time can use a context block to reference other VMScluster nodes. PSCANCTX$V_BUSY in PSCANCTX$L_FLAGS locks the context block. If the bit is set when the kernel thread attempts to acquire the context block, the kernel thread is placed in a wait state for resource RSN$_ASTWAIT until the context block is available. When the request referencing the context block completes, the kernel thread is reentered with an AST routine that clears PSCANCTX$V_BUSY. On return from the AST routine, the kernel thread reexecutes the test of the busy bit, acquires the context block, and continues from this point.

5. If $PROCESS_SCAN's caller requested information on a thread-by-thread rather than on a process-by-process basis, PSCANCTX$V_THREAD in PSCANCTX$L_FLAGS is set. In this case, EXE$GETJPI returns information for the next kernel thread each time the context block is used.

   Otherwise, if the bit is clear, EXE$GETJPI's actions for the current use of the context block are determined by the JPI$V_THREAD bit in JPI$_GETJPI_CONTROL_FLAGS. If $GETJPI's caller requested a search by kernel thread rather than by process, JPI$V_THREAD is set.
6. EXE$GETJPI calls the process scan routine EXE$PSCAN_NEXT_PID to obtain the next EPID, whether for a kernel thread or for a process, and to update the context block.
7. EXE$PSCAN_NEXT_PID, in module PROCESS_SCAN, tries to find a local process or kernel thread that matches the search criteria in the context block and that the $GETJPI requestor can access.

   To scan the processes on the local node, EXE$PSCAN_NEXT_PID steps through the PCB vector one process at a time. It updates the process index with each iteration, and the next request using the same context begins where the previous scan left off.

   However, to locate a process's kernel threads, it is necessary to step through the process's KTB vector. When the search is for a kernel thread rather than for a process, EXE$PSCAN_NEXT_PID checks whether the next sequence number is greater than the maximum number of threads per process. If the sequence number is higher, all possible process threads have been located. If so, or if no more valid entries remain in the process's KTB vector, the search returns to the PCB vector for

the next process. Otherwise it retrieves the next KTB from the KTB vector.

Thus, for multiple iterations, the initial iteration returns the process PCB, equivalent to the KTB of the initial kernel thread of the process, and each subsequent iteration returns the KTB of the process's next kernel thread. When the process has no more kernel threads, EXE$PSCAN_NEXT_PID locates the next process in the PCB vector.

The context block fields PSCANCTX$L_CUR_IPID, PSCANCTX$L_CUR_EPID, and PSCANCTX$L_NEXT_IPID track this local scan.

When EXE$PSCAN_NEXT_PID discovers a process or kernel thread that matches the search criteria, and to which the requestor has access, it returns the EPID to EXE$GETJPI, and EXE$GETJPI continues at step 8.

If it does not find a local process, EXE$PSCAN_NEXT_PID returns the CSID of the next node to search and the error status SS$_REMOTE_PROC. EXE$GETJPI passes control to CWPS$GETJPI_PSCAN, which continues the $GETJPI processing (see Section 6.2.4).

If it does not find a local process and the context block contains no more CSIDs to search, EXE$PSCAN_NEXT_PID returns the error status SS$_NOMOREPROC, which becomes the $GETJPI status return. EXE$GETJPI then deallocates the context block and stores the value $FFFFFFE0_{16}$ in the PID argument, ensuring that a subsequent call will fail immediately.

8. EXE$GETJPI calls EXE$NAM_TO_PCB to obtain the target PCB and KTB addresses and check privileges. As described in Section 6.1.3, EXE$NAM_TO_PCB determines whether the current kernel thread has the ability to obtain information about the target.

   If the target is not on the local node, EXE$NAM_TO_PCB returns the error status SS$_REMOTE_PROC. EXE$GETJPI passes control to CWPS$GETJPI, which continues the $GETJPI processing (see Section 6.2.4).

   If the target is on the local node, EXE$GETJPI continues with the steps that follow. These steps apply only to a target on the local node.
9. EXE$GETJPI checks for write access to the I/O status block (IOSB) and clears the IOSB, if one was specified.
10. It clears the event flag.
11. If AST notification was requested, EXE$GETJPI checks that the process has sufficient AST quota. If so, it charges for the AST; otherwise it returns the error status SS$_EXASTLM. Note that all kernel threads in a process share the process's AST quota, which is stored in the PCB.
12. EXE$GETJPI checks each item in the item list for the following conditions:

    - The buffer descriptor must be readable and the buffer writable.
    - The requested item must be a recognized one.
    - The return address length field, if specified, must be writable.

13. If these conditions are met, then the requested item can be retrieved. EXE$GETJPI moves all such information to the user-defined buffers for each corresponding item.

    Information known to be memory-resident, or information that the current kernel thread can fault into memory, can be retrieved directly:

    - All data about the current kernel thread, or about other kernel threads sharing the same process context, can be obtained directly. EXE$GETJPI can only return limited information about the swapper process, which has no control region or JIB.
    - PCB, JIB, KTB and ARB data about another process can be obtained directly without entering the context of the target process. These data structures reside in nonpaged pool and are allocated for the life of the thread, process, and job.

      Data from the PHD of another process can be obtained directly if the PHD is resident (the PCB$V_PHDRES bit in PCB$L_STS is set).

14. If no information remains to be gathered, then EXE$GETJPI returns to the requestor after performing the following actions:

    - Setting the specified event flag
    - Queuing AST notification, if it was requested
    - Writing status to an IOSB, if one was supplied

15. Information in the target process's control region can be retrieved only by executing in the context of the target process. Information stored in the target process's process header may not be available if the process is outswapped. To collect information from the control region or from an outswapped process header, EXE$GETJPI queues a special kernel mode AST to the target kernel thread, enabling EXE$GETJPI code to execute in the target context.

    The $GETJPI requestor can control this behavior through two $GETJPI control flags, JPI$V_NO_TARGET_INSWAP and JPI$V_NO_TARGET_AST.

    - If the requestor specifies JPI$V_NO_TARGET_INSWAP, EXE$GETJPI does not queue an AST to the target unless it is resident. Thus, EXE$GETJPI is unable to obtain any information about an outswapped process, but it can obtain information from the PHD and from the control region of a resident process.
    - If the requestor specifies JPI$V_NO_TARGET_AST, EXE$GETJPI never queues an AST to the target process. Thus, it returns data from a resident PHD but never from the control region.

    EXE$GETJPI does not queue the special kernel mode AST if the target PCB is the null PCB or the swapper PCB.

    Depending on the control flags, EXE$GETJPI allocates nonpaged pool for an extended AST control block (ACB) and an information buffer. It charges the pool against the requestor's JIB$L_BYTCNT quota and the

ACB against the requestor's PCB$L_ASTCNT. EXE$GETJPI initializes the normal ACB fields. It marks the ACB as thread-safe so that it will only be delivered to the kernel thread specified by the IPID. (AST delivery optimization for kernel threads can cause non-thread-safe ASTs to be delivered to whichever kernel thread currently holds the process's inner mode semaphore, as described in Chapter 5.)

In the extension, EXE$GETJPI stores descriptors of all the information that must be retrieved while executing in the context of the other process. It creates a buffer to receive the retrieved information for transmission to the requesting process.

16. EXE$GETJPI checks the status and state of the target process. If the target process is in any of the following states, information from it cannot be obtained:

    - It no longer exists.
    - Deletion or suspension is pending.
    - The process state is suspended (SUSP), suspended outswapped (SUSPO), or miscellaneous wait (MWAIT) (see Chapter 2).

    If the process is in any of these states, EXE$GETJPI deallocates the nonpaged pool and restores the quota charged. If the process no longer exists, EXE$GETJPI returns the error status SS$_NONEXPR to its requestor. For the other conditions, EXE$GETJPI's behavior is based on the $GETJPI control flag JPI$V_IGNORE_TARGET_STATUS. If the flag is set, EXE$GETJPI returns the status SS$_NORMAL to its requestor; otherwise it returns the error status SS$_SUSPENDED. Even in this case, EXE$GETJPI has already moved data from the PCB, JIB, and possibly the PHD into user-defined buffers (see step 13).

    Note that the completion mechanisms are all triggered if any error condition occurs. That is, the event flag is set, an AST (if requested) is queued, and an IOSB (if specified) is written with the failure status.

17. EXE$GETJPI queues the ACB to the target kernel thread with a priority increment class of PRI$_TICOM. If, however, the target kernel thread is computable (COM) or computable outswapped (COMO), queuing the AST does not result in a priority boost. (Chapter 2 discusses event reporting.) In that case, EXE$GETJPI boosts the target kernel thread's priority enough to make it equal to the priority of the requesting kernel thread (unless the requesting kernel thread is a real-time thread or its priority is lower than that of the target). The target priority boost ensures that even a low-priority target kernel thread will eventually return an answer to the requestor.
18. The asynchronous form of the system service returns to the requestor. The requestor can either wait for the information to be returned or continue processing. The synchronous form of the system service waits for the event flag associated with the request to be set and status to

be returned. Chapter *System Service Dispatching* gives details on synchronous and asynchronous system services.

### 6.2.4 Remote $GETJPI Support

The CWPS routines CWPS$GETJPI and CWPS$GETJPI_PSCAN, in module CWPS_GETJPI, dispatch $GETJPI requests to other VMScluster nodes and return status and item list information to the original requestor. EXE$GETJPI passes control to CWPS$GETJPI when no context block is associated with the request, merely an EPID or process name identifying a remote VMScluster node. It passes control to the alternative entry point CWPS$GETJPI_PSCAN when a context block exists.

CWPS$GETJPI performs the same argument list validation as EXE$GETJPI: checking the IOSB for write access and clearing it, clearing the specified event flag, checking and charging AST quota, and validating the item list.

If these checks succeed, CWPS$GETJPI allocates sufficient nonpaged pool to describe the $GETJPI request. It creates a variable-sized data structure with space for the context block, item list, return buffer, and an ACB. The $CWPSSRV macro defines the symbolic offsets for the fields in the structure header. The $CWPSJPI macro defines symbolic offsets for the fields in an extension for $GETJPI requests.

CWPS$GETJPI initializes this structure and stores its address in the context block, if one exists, at offset PSCANCTX$L_CWPSSRV. It calls a CWPS subroutine to transmit the request to the appropriate remote node using SCS.

On the remote node, a CWPS dispatch routine executing in system context receives the service request. It allocates a structure to describe the request, including the context block. It then queues a kernel mode AST to the CLUSTER_SERVER process, determining the procedure value of the AST procedure from the function to be performed; in this case, CWPS$SRCV_GETJPI_AST in module [SYSLOA]CWPS_SERVICE_RECV.

CWPS$SRCV_GETJPI_AST, executing in the context of the CLUSTER_SERVER process, builds a structure that mimics the PCB of the requesting process. If it did not receive a context block, it merely requests the $GETJPI system service on behalf of the original requestor. Otherwise, it inserts the context block into the PHD queue in the CLUSTER_SERVER process and calls the process scan routines EXE$PSCAN_LOCKCTX and EXE$PSCAN_NEXT_PID as EXE$GETJPI does. These locate the EPID of the next process matching the search context. CWPS$SRCV_GETJPI_AST then requests the $GETJPI service with the explicit EPID of a local process, specifying a completion AST procedure. EXE$GETJPI follows the steps described in Section 6.2.3 for a local process.

When the $GETJPI request completes, the completion AST procedure passes control to another CWPS routine to return status and data to the originating node using SCS.

When the response arrives from the remote node, a cleanup routine tests the status returned from the remote node. On a successful return, it copies the returned data to the $GETJPI requestor's buffer area after suitable accessibility checks, updates the context block, and clears the busy flag. It sets the event flag, queues an AST (if requested), and returns the status in the IOSB, if one is specified.

### 6.2.5 $GETJPI Special Kernel Mode ASTs

To obtain information about a target process on the local node, EXESGETJPI must sometimes queue a special kernel mode AST to the target. From either the context of the requesting process (if the requestor is local) or the context of the CLUSTER_SERVER process, EXESGETJPI queues this AST when the required information resides in an outswapped PHD or in the control region of the target process.

The special kernel mode AST routine executes in the context of the target process to access the information. Once the AST has obtained the information, it queues another special kernel mode AST to the requesting process or the CLUSTER_SERVER process to pass the information back to the service requestor.

A summary of the operations performed by these two special kernel mode AST routines follows:

1. The first special kernel mode AST routine runs when the target kernel thread is placed into execution. It examines the extended ACB to determine the requested information and stores it in the associated system buffer. It reformats the extended ACB to deliver a second special kernel mode AST, this time to the requesting process or the CLUSTER_SERVER process. It queues the extended ACB to the requesting process if it still exists and is not marked for deletion. (The CLUSTER_SERVER process cannot be deleted or suspended.) Otherwise, it deallocates the ACB and returns.
2. The second kernel mode AST routine executes in the context of the requesting process or CLUSTER_SERVER process. If the PHD image counter has changed since the service was requested, then the requesting image has been run down. In this case, the AST routine deallocates the nonpaged pool, restores the JIB$L_BYTCNT quota, and returns.
3. If the image counter in the PHD agrees with the image counter in the extended ACB, the special kernel mode AST routine copies the retrieved data from the system buffer into the user-defined buffers.

   Note that the asynchronous nature of this aspect of the system service requires that the IOSB and all data buffers be probed again for write accessibility. This check ensures that the original requestor of $GETJPI has not altered the IOSB and data buffer protection in the interval between the $GETJPI request and the delivery of the return special kernel mode AST.

4. The event flag is set and the IOSB is written if it was specified.
5. If a completion AST was requested, the extended ACB is reused to queue an AST to the requesting process in the requestor's access mode. Otherwise, the ACB is deallocated to nonpaged pool.

   The CLUSTER_SERVER process always specifies CWPS$SRCV_GETJPI_SRV_AST as its completion AST procedure when requesting the $GETJPI system service. Therefore, for a remote request, the ACB is always reused.

### 6.2.6 Traditional Wildcard Support in $GETJPI

In addition to the wildcard search available through the $PROCESS_SCAN system service, the executive preserves the traditional $GETJPI wildcard behavior. The $GETJPI system service provides the ability to obtain information about all processes on the local node. An image requests this feature by passing −1 as the PID argument to the $GETJPI system service. An internal routine in EXE$GETJPI searches the PCB vector for the first slot containing a valid PCB and passes information back to the requestor about the associated process.

EXE$GETJPI alters the process index field of the requestor's PID argument to contain the process index of the target process. When the $GETJPI service is requested again, the negative sequence number (in the high-order word of the process ID) indicates that a wildcard operation is in progress, and the positive process index indicates the offset in the PCB vector where the search should continue. The PID process index field is updated in this manner for each iteration, and the wildcard operation continues process by process.

The image continues to request the $GETJPI service until the error status SS$_NOMOREPROC is returned, indicating that the PCB vector search routine has reached the end of the PCB vector. EXE$GETJPI then stores the value $FFFFFFE0_{16}$ in the PID argument, ensuring that a subsequent call will fail immediately.

The traditional $GETJPI wildcard behavior also allows wildcarding through all the kernel threads of a single process. When the JPI$V_THREAD flag is set in the JPI$_GETJPI_CONTROL_FLAGS item code, $GETJPI returns information about the process's initial kernel thread on the first iteration. It alters the requestor's PID argument to contain the EPID of the target kernel thread, from which both the target process PCB and the initial kernel thread KTB can be located. It sets bit 31 of the PID argument, making it a negative value. It also sets bit 30, PCB$V_EPID_WILD_ONE_PROCESS, denoting that the wildcard operation affects the kernel threads of a single process.

When the $GETJPI service is requested again, the negative value indicates that a wildcard operation is in progress, PCB$V_EPID_WILD_ONE_

PROCESS indicates that the target is a kernel thread, and the EPID identifies both the target PCB and the thread offset in the KTB vector where the search should continue. The PID argument is updated in this manner for each iteration, and the wildcard operation continues kernel thread by kernel thread.

The image continues to request the $GETJPI service until the error status SS$_NOMORETHREAD is returned, indicating that EXE$GETJPI has reached the end of the process's KTB vector, that is, no more kernel threads remain for the process. EXE$GETJPI then stores the value $FFFFFFE1_{16}$ in the PID argument, ensuring that a subsequent call will fail immediately.

Chapter 2 provides more information on the PCB vector, and Chapter 5 describes the KTB vector. Note that the user image will not work correctly if it alters the value of the PID argument between $GETJPI requests. The *OpenVMS System Services Reference Manual* contains a sample program using $GETJPI wildcards.

## 6.3 SYSTEM SERVICES AFFECTING PROCESS COMPUTABILITY

The controlling process in a multiprocess application typically creates other processes to perform designated work. When these processes have completed their work, the controlling process may delete them or place them into a wait state in anticipation of additional work. Chapter 7 describes process creation, and Chapter 10, process deletion.

In an application built with a user mode thread manager, multiple user mode threads can be scheduled on a kernel thread of the process. OpenVMS Alpha Version 7.0 provides an upcall mechanism by which the executive notifies the user mode thread manager of events that affect the computability of user mode threads. When a user mode thread manager registers itself in a process, it supplies the address of its upcall table, a table listing entry points for various types of upcall events. Before placing a kernel thread into a hibernate wait, for example, the OpenVMS executive checks whether there is a user mode thread manager hibernate entry point. If an entry point for that upcall type exists, OpenVMS calls the thread manager. The thread manager's actions depend upon the event type. For instance, one possible action is to schedule some other user mode thread on the kernel thread to replace a stalling user mode thread. Chapter 5 describes the upcall mechanism and the actions taken for various events. The following sections describe the conditions under which upcalls are made for various process control system services.

Hibernation and suspension are the two different ways in which a process can temporarily stall execution. The system services Hibernate ($HIBER) and Suspend Process ($SUSPND) implement hibernation and suspension. The associated services Wake Process ($WAKE), Schedule Wakeup ($SCHDWK), and Resume Process ($RESUME) cause execution to recommence.

### 6.3.1 Hibernate/Wake

**6.3.1.1 Hibernate System Service.** A kernel thread requests the $HIBER service to place itself into hibernation; it cannot put another of its process's kernel threads into the hibernating (HIB) state, and it cannot put another process into the HIB state. The $HIBER system service procedure is EXE$HIBER, in module SYSPCNTRL. EXE$HIBER runs in the mode of its caller. It exists primarily as a jacket for the internal kernel mode system service $HIBER_INT. EXE$HIBER performs the following:

- If the calling kernel thread is executing in user mode and CTL$GL_MULTITHREAD is nonzero, EXE$HIBER checks for the presence of a user mode thread manager. If one is registered and has an entry for hibernation in its upcall table, EXE$HIBER calls the user mode thread manager to notify it of the state change.

  The thread manager stalls the current user mode thread and potentially schedules some other user mode thread on this kernel thread. Control does not return to EXE$HIBER until the stalled user mode thread is rescheduled by the thread manager, generally after a $WAKE. Chapter 5 describes threads and the upcall mechanism.
- Otherwise, for single-threaded processes and for kernel threads executing in an elevated access mode or without a user mode thread manager, EXE$HIBER requests the $HIBER_INT system service immediately.

**6.3.1.2 Internal Hibernate System Service.** The $HIBER_INT system service procedure is EXE$HIBER_INT, in module SYSPCNTRL. EXE$HIBER_INT is a thread-safe system service that runs in kernel mode. It performs the following:

1. EXE$HIBER_INT obtains the address of the calling kernel thread's KTB.
2. It tests whether proactive memory reclamation from processes that wake periodically has been enabled and is currently active, and whether this process is potentially subject to working set reduction.

   To be subject to working set reduction, all kernel threads of a process must be at a priority below the real-time priority range, and each kernel thread must be in a scheduling state of local event flag (LEF) wait, common event flag (CEF) wait, or HIB.
3. EXE$HIBER_INT acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
4. It tests and clears the state of the wake pending flag, KTB$V_WAKEPEN in KTB$L_STS.
5. If the flag was set, a wake request has occurred. EXE$HIBER_INT releases the spinlock and returns to its requestor at IPL 0.
6. If the process is potentially subject to working set reduction, EXE$HIBER_INT acquires the MMG spinlock at IPL$_MMG. It calls EXE$CHK_WAIT_BHVR, in module RSE, to reduce the working set if appropriate.

Chapter 2 describes the routine and this form of memory reclamation in more detail. EXE$HIBER_INT then releases the MMG spinlock.

7. To actually hibernate the kernel thread, EXE$HIBER_INT calls SCH$WAIT_PROC, in module SCHED_ROUTINES, to place the kernel thread's KTB into the hibernate wait queue. It then returns the status SS$_WAIT_CALLERS_MODE to the system service dispatcher, causing the kernel thread to be taken out of execution.

   The system service dispatcher alters the saved program counter (PC) to contain the address of the beginning of the system service transfer routine. Thus, if the kernel thread receives an AST while hibernating, it reexecutes EXE$HIBER_INT upon completion of the AST routine. Since EXE$HIBER_INT tests the wake pending flag, a hibernating kernel thread is easily awakened if an AST procedure requests the $WAKE service. Chapter *System Service Dispatching* describes the actions of the system service dispatcher.

6.3.1.3 **Wake System Services.** $HIBER's complementary services are $WAKE and $SCHDWK, which remove a single kernel thread or all of a process's kernel threads from hibernation. To awaken any or all of the kernel threads sharing its process context, a kernel thread can request $WAKE from an AST procedure or schedule a wake through $SCHDWK. Alternatively, a kernel thread in a different process context with the ability to affect the hibernating kernel thread can request $WAKE or $SCHDWK on the hibernating kernel thread's behalf.

The $WAKE system service procedure, EXE$WAKE in module SYSPCNTRL, runs in kernel mode. It calls EXE$NAM_TO_PCB, described in Section 6.1.3.

If the process is remote, EXE$WAKE branches to CWPS$P_CNTRL, in module SYSPCNTRL. Section 6.1.4 summarizes the result.

If the process is local, EXE$WAKE performs the following:

- If the process has only a single kernel thread, EXE$WAKE calls SCH$WAKE in module RSE.
- If the process has multiple kernel threads but the PID argument was specified, EXE$WAKE calls SCH$WAKE to remove from hibernation the specific kernel thread identified by the PID argument.
- If the process has multiple kernel threads and the PID argument was not specified, EXE$WAKE attempts to remove all of the process's kernel threads from hibernation. It calls SCH$WAKE for the initial kernel thread. For each other occupied slot in the KTB vector, EXE$WAKE then calls SCH$WAKE2, also in module RSE.

SCH$WAKE tests PCB$L_MULTITHREAD to see whether multithreading is enabled for the process. If so, it checks for the presence of a user mode thread manager. If one is registered and has an entry for wake in its upcall

table, and the system service was requested from user mode, SCH$WAKE calls EXE$INFORM_TM_AST (see Chapter 5), in module TM_SUPPORT, to queue an AST to this kernel thread.

Later, after the kernel thread is awakened, the AST will be delivered. In user mode in the kernel thread's context, SCH$WAKE notifies the user mode thread manager of the state change. The thread manager reschedules a now runnable user mode thread in the kernel thread.

SCH$WAKE then joins SCH$WAKE2, which sets the wake pending flag, KTB$V_WAKEPEN. It then reports the awakening event to the scheduler routine SCH$REPORT_EVENT, specifying the priority boost class PRI$_RESAVL for the awakening kernel thread. SCH$REPORT_EVENT removes the kernel thread from the HIB or HIBO queue and places it into the COM or COMO queue corresponding to its updated priority.

The next time the kernel thread is scheduled at non-AST level, EXE$HIBER reexecutes because of the altered PC. Since SCH$WAKE set the wake pending flag, EXE$HIBER clears the flag and returns immediately. Note that if a kernel thread is awakened from any state other than HIB or HIBO, the net result is to leave the wake pending flag set with no other change in the scheduling state.

Chapter 2 provides further details on SCH$REPORT_EVENT, priority boosts, and process state queues, and Chapter *Time Support* describes the $SCHDWK system service.

### 6.3.2 Suspend/Resume

Because one process can suspend other processes within the VMScluster system, the implementation of process suspension is more complicated than that of hibernation. The executive's scheduling philosophy illustrated in Figure 2.12 assumes that processes enter various wait states from the state of being the current process and in no other way. This assumption requires that the process being suspended (the target) become the current process on some CPU, possibly replacing the requestor of the $SUSPND system service.

The $SUSPND and $RESUME system services affect all kernel threads in a process. Therefore, each kernel thread must become current in order for the process to be suspended. To accommodate this scheduling constraint, $SUSPND issues a kernel or supervisor mode AST to each kernel thread. AST execution ensures that each kernel thread is first made current before being placed into the SUSP scheduling state.

While each kernel thread must be removed from execution individually, the overall state of the process (suspended or not) is reflected in the bits PCB$V_SUSPEN and PCB$V_SOFTSUSP of PCB$L_STS rather than in each KTB.

A process can be suspended either in kernel mode or in supervisor mode through the $SUSPND system service. Kernel mode suspension is called

hard suspension; supervisor mode suspension, the default, is called soft suspension. No ASTs can be delivered to a hard-suspended process, whereas executive and kernel mode ASTs can be delivered to a soft-suspended process. The $RESUME system service makes a suspended process computable.

6.3.2.1 **Process Suspension.** For a process with a single kernel thread, process suspension occurs in two phases. In the first phase, the $SUSPND system service procedure, EXE$SUSPND, queues a supervisor mode or kernel mode AST procedure to the process that is to be suspended. In the second phase, the AST procedure executes in the target process's context.

For a process with multiple kernel threads, a supervisor mode or kernel mode AST procedure must be queued to each kernel thread.

The queuing of the ASTs must occur in the context of the process itself, because the kernel thread queuing the ASTs must hold the inner mode semaphore, a process-private synchronization mechanism, to block the creation of new kernel threads. Chapter 5 gives information on kernel threads, the inner mode semaphore, and AST delivery to kernel threads.

A kernel thread suspending its own process acquires its process's inner mode semaphore via the system service dispatcher when it requests the $SUSPND system service. Therefore, EXE$SUSPND can safely queue a supervisor mode or kernel mode AST procedure to each kernel thread.

However, to suspend a different process, EXE$SUSPND must queue an intermediary kernel mode AST procedure to the target process. Executing in the target process's context and holding that process's inner mode semaphore, this intermediary AST procedure queues a supervisor mode or kernel mode AST procedure to each kernel thread.

Finally, the supervisor or kernel mode AST procedure executes in each target kernel thread's context.

6.3.2.1.1 **EXE$SUSPND.** EXE$SUSPND performs the following steps in kernel mode:

1. EXE$SUSPND checks for the presence of its FLAGS argument. If the low bit of the FLAGS argument is clear or the argument is not specified, the request is for supervisor mode suspension.
2. If the request is for kernel mode suspension, EXE$SUSPND checks whether it was requested from either kernel or executive mode; if not, it returns the error status SS$_NOPRIV.
3. EXE$SUSPND calls EXE$NAM_TO_PCB to identify the target process and to perform access checking.
4. If the target process is not local, EXE$NAM_TO_PCB returns the error status SS$_REMOTE_PROC. EXE$SUSPND passes the request to a CWPS routine for transmission to a remote VMScluster node. If the local process has appropriate access to the remote target process, a CWPS routine on the remote node eventually executes the $SUSPND request

in the context of the CLUSTER_SERVER process. It transmits status to a CWPS receiver on the requesting node, which reenters the context of the requesting process via an AST to return the status to the user image.

5. Otherwise, if the target process is local, EXE$NAM_TO_PCB returns holding the SCHED spinlock. EXE$SUSPND continues with the steps that follow. (Exit paths from EXE$SUSPND must release this spinlock and lower IPL.)
6. EXE$SUSPND checks the delete pending bit PCB$V_DELPEN, in PCB$L_STS, in the PCB of the target process. If the process is marked for deletion, it returns the error status SS$_NONEXPR.
7. It checks the bit PCB$V_NOSUSPEND in PCB$L_STS. If EXE$SUSPND cannot safely suspend the process, it returns the error status SS$_NOSUSPEND.
8. It tests and clears PCB$V_RESPEN, the resume pending bit in PCB$L_STS. If the $RESUME system service set this bit before the process was actually suspended, $SUSPND cancels the suspension.
9. It tests PCB$V_SUSPEN, the suspend pending bit in PCB$L_STS. If suspension is pending, EXE$SUSPND tests the bit PCB$V_SOFTSUSP. If the pending suspension is supervisor mode, PCB$V_SOFTSUSP is set; otherwise, a kernel mode suspension is pending.

   - If its $SUSPND request is for supervisor mode suspension and either a kernel mode suspension or a supervisor mode suspension is pending, EXE$SUSPND returns the status SS$_NORMAL.
   - If its $SUSPND request is for kernel mode suspension and a kernel mode suspension is pending, EXE$SUSPND returns the status SS$_NORMAL.
   - If its $SUSPND request is for kernel mode suspension and a supervisor mode suspension is pending, EXE$SUSPND clears PCB$V_SOFTSUSP. Kernel mode suspension preempts the supervisor mode suspension.
   - If no suspension is pending, EXE$SUSPND sets PCB$V_SUSPEN, the suspend pending bit in PCB$L_STS. If its $SUSPND request is for supervisor mode suspension, it also sets the bit PCB$V_SOFTSUSP.

10. EXE$SUSPND examines PCB$L_KT_COUNT; if it contains a value greater than 1, the process has multiple kernel threads.
11. EXE$SUSPND compares its target PCB and the current process's PCB to determine whether its caller is suspending itself.
12. If EXE$SUSPND's caller is not suspending itself, and the target process has multiple kernel threads, EXE$SUSPND queues a kernel mode AST to the target process, specifying the intermediary AST procedure. It passes the suspension mode as the AST parameter. Because EXE$SUSPND does not declare the intermediary AST procedure to be thread-safe, the AST will execute on a kernel thread holding the target process's inner mode semaphore.

The intermediary kernel mode AST procedure will queue the actual kernel mode or supervisor mode suspension ASTs to each of the process's kernel threads.

Through the normal scheduling selection process, the intermediary kernel AST is eventually delivered to one of the target process's kernel threads.

13. Otherwise, if EXE$SUSPND's caller is suspending itself, EXE$SUSPND examines PCB$L_KTBVEC, which locates a KTB for each kernel thread. In the case of a single-threaded process, PCB$L_KTBVEC locates the PCB. EXE$SUSPND allocates an ACB for each KTB and queues either a kernel mode AST or a supervisor mode AST to every kernel thread, based on the suspension mode.

    If queuing a kernel mode AST, EXE$SUSPND sets the ACB$V_THREAD_SAFE bit to ensure that the AST is delivered to the desired kernel thread rather than to whichever kernel thread holds the inner mode semaphore (see Chapter 5).

    Through the normal scheduling selection process, each target kernel thread eventually executes the kernel or supervisor mode AST procedure.

6.3.2.1.2 **Intermediary Kernel Mode AST Procedure.** The kernel mode AST procedure SUSPND_INTERIM_AST, in module SYSPCNTRL, executes in the context of one of the target process's kernel threads, holding the inner mode semaphore. It obtains the current PCB address, acquires the SCHED spinlock, and locates its parameter, the suspension mode. (Exit paths must release this spinlock and lower IPL.)

Before it queued the intermediary AST, EXE$SUSPND set the bit PCB$V_SUSPEN in PCB$L_STS and PCB$V_SOFTSUSP if the suspension request was for supervisor mode. SUSPND_INTERIM_AST now tests whether these flag bits have been altered since then by a $RESUME request or a $SUSPND request for a different mode than that specified in SUSPND_INTERIM_AST's parameter.

- If SUSPND_INTERIM_AST's parameter indicates kernel mode suspension but PCB$V_SUSPEN and PCB$V_SOFTSUSP indicate no suspension or supervisor mode suspension, SUSPND_INTERIM_AST exits without queuing an AST.
- If SUSPND_INTERIM_AST's parameter indicates supervisor mode suspension but PCB$V_SUSPEN and PCB$V_SOFTSUSP indicate no suspension or kernel mode suspension, SUSPND_INTERIM_AST exits without queuing an AST.

Otherwise, SUSPND_INTERIM_AST examines PCB$L_KTBVEC, which locates a KTB for each kernel thread. It allocates an ACB for each KTB and queues either a kernel mode AST or a supervisor mode AST to every kernel

thread, based on the suspension mode. If queuing a kernel mode AST, it sets the ACB$V_THREAD_SAFE bit to ensure that the AST is delivered to the desired kernel thread rather than to whichever kernel thread holds the inner mode semaphore.

If a failure occurs while allocating nonpaged pool for the ACBs, SUSPND_INTERIM_AST normally calls SCH$RESOURCE_WAIT_PS to place the kernel thread into a resource MWAIT state for nonpaged dynamic memory (see Chapter 2). However, if the process has disabled resource waits, SUSPND_INTERIM_AST cannot continue and the process cannot be suspended. It deallocates any ACBs acquired, clears PCB$V_SUSPEN and PCB$V_SOFTSUSP, and returns to its caller. No error is returned from this failure; EXE$SUSPND returned SS$_NORMAL to the original caller when the intermediary AST was queued.

6.3.2.1.3 **Kernel Mode AST Procedure.** The kernel mode AST procedure HARD_SUSPND_AST, in module SYSPCNTRL, executes in the context of the target kernel thread. Since the ACB$V_THREAD_SAFE bit was set, it does not hold the target process's inner mode semaphore. HARD_SUSPND_AST obtains the current PCB address and acquires the SCHED spinlock. (Exit paths must release this spinlock and lower IPL.)

Before this AST was queued, EXE$SUSPND ensured that the process was suspendable and not marked for deletion by testing the bits PCB$V_DELPEN and PCB$V_NOSUSPEND in PCB$L_STS. HARD_SUSPND_AST now tests the bits again, exiting if either is set.

Before this AST was queued, EXE$SUSPND set the bit PCB$V_SUSPEN in PCB$L_STS. HARD_SUSPND_AST now tests whether this flag bit, or PCB$V_SOFTSUSP, has been altered since that time by a $RESUME request possibly followed by a $SUSPND request for supervisor mode. If PCB$V_SUSPEN is clear, or both it and PCB$V_SOFTSUSP are set, HARD_SUSPND_AST exits.

HARD_SUSPND_AST obtains the current KTB. It tests PCB$L_DPC for outstanding file system activity by the Files-11 Extended QIO Processor (XQP). If it is nonzero, a file system request is active on behalf of this process and a kernel mode AST will be delivered on completion. Such an AST would be undeliverable to a process suspended in kernel mode, so HARD_SUSPND_AST calls SCH$RESET_ASTSR to recompute the AST level, then calls SCH$RESOURCE_WAIT_PS to place the kernel thread in a resource MWAIT state for AST delivery (see Chapter 2). When the AST is delivered, HARD_SUSPND_AST restarts its checks from the beginning.

If no file system activity is in progress, HARD_SUSPND_AST calls SCH$WAIT_PROC to place the kernel thread on the SUSP wait queue, then SCH$WAIT_KERNEL_MODE_PS to remove the thread from execution and schedule the next available kernel thread. The thread's saved PC is an ad-

dress in HARD_SUSPND_AST, and the saved processor status (PS) indicates kernel mode and IPL 0.

ASTs can be queued to a process suspended in kernel mode but they cannot be delivered. When an AST is queued to a process suspended in kernel mode, SCH$REPORT_EVENT ignores the AST event. Only the $RESUME system service or the $DELPRC system service can cause a process suspended in kernel mode to continue with execution. At that time, HARD_SUSPND_AST is reentered. It reexecutes the check of PCB$V_SUSPEN in PCB$L_STS and discovers it to be clear, causing the kernel thread to return successfully.

6.3.2.1.4 **Supervisor Mode AST Procedure.** The supervisor mode AST procedure, SOFT_SUSPND_AST in module SYSPCNTRL, executes in the context of the target kernel thread. Its only action is to request the internal form of the soft suspend system service, $SOFT_SUSPND_INT.

6.3.2.1.5 **Soft Suspend System Service.** The soft suspend system service, $SOFT_SUSPND_INT, is designed to be requested from the supervisor mode AST routine SOFT_SUSPND_AST. The system service procedure, EXE$SOFT_SUSPND_INT in module SYSPCNTRL, executes in kernel mode in the context of the target kernel thread, holding the target process's inner mode semaphore. EXE$SOFT_SUSPND_INT obtains the current PCB address and acquires the SCHED spinlock. (Exit paths must release this spinlock and lower IPL.)

Before this AST was queued, EXE$SUSPND tested the bits PCB$V_DELPEN and PCB$V_NOSUSPEND in PCBSL_STS to ensure that the process was suspendable and not marked for deletion. EXE$SOFT_SUSPND_INT now tests the bits again, exiting if either is set.

Before this AST was queued, EXE$SUSPND set the bit PCB$V_SUSPEN in PCB$L_STS. EXE$SOFT_SUSPND_INT now tests whether this flag bit or PCB$V_SOFTSUSP has been altered since that time by a $RESUME request or a $SUSPND request. Unless a supervisor mode suspension is still desired, EXE$SOFT_SUSPND_INT exits.

EXE$SOFT_SUSPND_INT calls SCH$WAIT_PROC to place the kernel thread on the SUSP wait queue. It then returns the status SS$_WAIT_CALLERS_MODE to the system service dispatcher, causing the process to be taken out of execution.

While a process is suspended in supervisor mode, its saved PC contains the address of the beginning of the EXE$SOFT_SUSPND_INT system service transfer routine. Its saved PS indicates supervisor mode and the process's supervisor mode AST active bit is set, blocking delivery of user and supervisor mode ASTs.

The queuing of an AST makes the process computable. When the process is placed into execution, a kernel or executive mode AST can be executed,

but a user or supervisor mode AST cannot; the AST control block is queued and the interrupt is dismissed.

When the interrupt is dismissed, the `CALL_PAL REI` instruction is executed. This causes the $SOFT_SUSPND_INT system service to be reexecuted. EXE$SOFT_SUSPND_INT repeats the test that suspended the process. If PCB$V_SUSPEN or PCB$V_SOFTSUSP is still set, the process is once more suspended.

6.3.2.2 **$RESUME System Service.** Just as the $SUSPND system service suspends all of a process's kernel threads, the $RESUME system service must resume each one.

The $RESUME system service procedure, EXE$RESUME in module SYSPCNTRL, runs in kernel mode. EXE$RESUME calls EXE$NAM_TO_PCB, obtaining the current PCB address and acquiring the SCHED spinlock. (Exit paths release this spinlock and lower IPL.) For an accessible process on the local system, EXE$RESUME tests the bit PCB$V_SUSPEN in PCB$L_STS to see whether the process is actually suspended. If not, it sets the resume pending flag PCB$V_RESPEN in the target process PCB and exits. The next $SUSPND system service request for the process will return immediately.

If the process is actually suspended, EXE$RESUME clears PCB$V_SUSPEN and PCB$V_SOFTSUSP. For each kernel thread, it then reports a resume event by calling SCH$REPORT_EVENT. As with all other scheduling events, this report may result in a rescheduling interrupt request, a request to wake the swapper process, or nothing at all.

If the target process is not local, the $RESUME request is passed to a CWPS routine for transmission to a remote VMScluster node. If the local process has appropriate access to the remote target process, a CWPS routine on the remote node eventually executes the $RESUME request in the context of the CLUSTER_SERVER process. It transmits status to a CWPS receiver on the requesting node, which reenters the context of the requesting process via an AST to return the status to the user image.

### 6.3.3 Exit and Forced Exit

The Exit ($EXIT) system service terminates the currently executing image. If the process is executing a single image without a command language interpreter, image exit usually results in process deletion. A detailed discussion of the $EXIT system service is given in Chapter 8.

The Force Exit ($FORCEX) system service enables a process to force a target process to request the $EXIT system service. The system service procedure EXE$FORCEX, in module SYSFORCEX, runs in kernel mode. It locates the process through EXE$NAM_TO_PCB.

For a local process, EXE$FORCEX performs the following:

1. If PCB$V_NODELET in PCB$L_STS is set, the process cannot be deleted. EXE$FORCEX simply returns a success status.
2. Otherwise it calls EXE$CHECK_UPCALL to see whether a user mode thread manager exists and has an upcall entry point for forced exit notification.

   If so, EXE$FORCEX allocates an ACB and calls EXE$INFORM_TM_AST, in module TM_SUPPORT, with the IPID of the initial kernel thread of the target process. EXE$INFORM_TM_AST queues an AST to this kernel thread so that the upcall is performed from the kernel thread's context.
3. Otherwise, for a process with only one kernel thread, a multithreaded process with no user mode thread manager, or a multithreaded process whose user mode thread manager does not provide a forced exit notification upcall, EXE$FORCEX checks whether the force exit pending flag, PCB$V_FORCPEN in PCB$L_STS, is set. If so, an exit is already in progress and EXE$FORCEX exits.

   If PCB$V_FORCPEN is clear, EXE$FORCEX sets it and and queues a user mode AST to the initial kernel thread of the target process.
4. This AST procedure, executing in user mode, tests whether CTL$GQ_EXIT_THREAD is nonzero. If so, an exit is already in progress and the AST procedure exits. Otherwise it clears the AST active flag by requesting the Clear AST ($CLRAST) system service. It then requests the $EXIT system service. Note that the AST procedure cannot execute if the target process has disabled AST delivery.

   Chapter 5 provides more information on user mode thread managers and upcalls. Chapter *ASTs* provides more information on ASTs and the $CLRAST system service.
5. The $EXIT request executes in the context of the target process. Execution proceeds as if the target process had requested the system service itself.

If EXE$FORCEX receives the error status SS$_REMOTE_PROC from EXE$NAM_TO_PCB, the target process is not local. EXE$FORCEX passes the request to a CWPS routine for transmission to a remote VMScluster node. If the local process has appropriate access to the remote target process, a CWPS routine on the remote node eventually executes the $FORCEX request in the context of the CLUSTER_SERVER process, performing the steps described in Section 6.1.4. The remote CWPS routine transmits status to a CWPS receiver on the requesting node, which reenters the context of the requesting process via an AST to return the status to the user image.

### 6.3.4 Process Affinity and Capability System Services

The process affinity and capability system services allow a kernel thread to affect scheduling policy for itself and others.

Affinity is a requirement that a kernel thread execute on a particular CPU or a subset of CPUs in an SMP system. The $PROCESS_AFFINITY and SSET_IMPLICIT_AFFINITY system services allow a kernel thread to retrieve or alter its own affinity requirements and that of any local process for which it has sufficient privilege.

A capability is a CPU attribute that a kernel thread requires in order to execute. User-defined capabilities, established by the Modify CPU User Capabilities ($CPU_CAPABILITIES) system service, have application-dependent meanings. The $PROCESS_CAPABILITIES system service allows a kernel thread to retrieve or alter its own user capability set or that of any local process for which it has sufficient privilege.

Since affinities and capabilities are only relevant in the context of an SMP system, these system services provide no interface to CWPS. All use EXE$NAM_TO_PCB to identify the target kernel thread, and thus follow its rules for processing the PID and PRCNAM arguments and for verifying privilege. Chapter 2 provides more detail on explicit and implicit affinity, CPU capacity, the KTB fields affected, and the actions of the scheduler in response to these system service requests.

6.3.4.1 **Modify Process Affinity System Service.** The $PROCESS_AFFINITY system service is a kernel mode system service that allows a kernel thread to examine or modify its current or permanent CPU affinity set, that is, the set of CPUs on which the kernel thread can execute. It also allows a kernel thread with appropriate privileges to examine or modify the current or permanent CPU affinity set of other local processes.

A kernel thread needs no privilege to retrieve its own affinity set or that of other kernel threads sharing the same process context or UIC group. It must have the ALTPRI privilege to modify its own affinity set, plus either GROUP or WORLD to affect other local processes.

6.3.4.2 **Modify Process Implicit Affinity System Service.** The $SET_IMPLICIT_AFFINITY system service allows a kernel thread to enable or disable implicit affinity for itself or for any other kernel thread that it has sufficient privilege to affect. It also allows a sufficiently privileged kernel thread to enable or disable the systemwide default implicit affinity, stored in the systemwide default capability mask, SCH$GL_DEFAULT_PROCESS_CAP.

Currently, enabling implicit affinity for a kernel thread introduces a scheduling bias; the executive will attempt to schedule the kernel thread on a CPU where its address translations and memory references are more likely to be in cache.

A kernel thread needs no privileges to retrieve its own implicit affinity state or that of other kernel threads sharing the same process context or UIC group. It must have the ALTPRI privilege to modify its own implicit affinity state, plus either GROUP or WORLD to affect other processes.

6.3.4.3 **Modify Process User Capabilities System Service.** The $PROCESS_CAPABILITIES system service allows a kernel thread to retrieve or alter its own current user capability set, the capability set of any other kernel thread sharing its process context, and the capability set of any kernel thread to which it has access privileges. It also allows a sufficiently privileged kernel thread to retrieve or alter SCH$GL_DEFAULT_PROCESS_CAP, the systemwide default process capability initialization field. This field determines the default capabilities assigned to all newly created kernel threads on the system.

A kernel thread needs no privileges to retrieve its own capability mask or that of other kernel threads sharing the same process context or UIC group. It must have the ALTPRI privilege to modify a capability mask, and either GROUP or WORLD to affect other processes.

## 6.4 MISCELLANEOUS PROCESS ATTRIBUTE CHANGES

Several system services allow a process to alter its characteristics, such as its response to resource allocation failures, its priority, and its process name. Some of these changes (such as priority elevation or swap disabling) require privilege. The Set Priority ($SETPRI) system service is the only service described in this section that a kernel thread can issue for a target other than itself.

### 6.4.1 Set Priority

The $SETPRI system service allows a kernel thread to alter its own priority or the priority of other kernel threads within the VMScluster system, limited by the privilege checks in EXE$NAM_TO_PCB (see Section 6.1.3). A kernel thread in a process with the ALTPRI privilege can change priority to any value between 0 and 64. A kernel thread in a process without this privilege is restricted to the range between 0 and the authorized base priority of its target kernel thread (PCB$L_AUTHPRI, KTB$L_AUTHPRI) or the current base priority of its target kernel thread (PCB$L_PRIB, KTB$L_PRIB), whichever is higher.

The system service procedure, EXE$SETPRI in module SYSSETPRI, runs in kernel mode. It locates the target kernel thread's KTB by calling EXE$NAM_TO_PCB. When only the PRCNAM argument is supplied, the target kernel thread returned by EXE$NAM_TO_PCB is the initial kernel thread of the target process.

If the target kernel thread is not local, EXE$NAM_TO_PCB returns the error status SS$_REMOTE_PROC. EXE$SETPRI passes the request to a CWPS routine for transmission to a remote VMScluster node. If the local process has appropriate access to the remote target process, a CWPS routine on the remote node eventually executes the $SETPRI request in the context of the

CLUSTER_SERVER process, performing the steps described in the following paragraphs. The remote CWPS routine transmits status to a CWPS receiver on the requesting node, which reenters the context of the requesting process via an AST to return the status to the user image.

For a local kernel thread, EXE$SETPRI changes the base priority in the KTB at offsets KTB$L_PRIBSAV and KTB$L_PRIB and the saved base priority at offset KTB$L_PRISAV. For a target kernel thread at elevated priority with a mutex locked, EXE$SETPRI only alters KTB$L_PRIBSAV and KTB$L_PRISAV. When the kernel thread unlocks the mutex, its priority values are restored from these fields. Chapter 2 provides further information on these KTB fields.

If the target kernel thread is currently executing, EXE$SETPRI calls SCH$CHANGE_CUR_PRIORITY, in module RSE, to alter its current priority, stored in offset KTB$L_PRI, and other priority-related systemwide data structures. Chapter 2 describes the operation of SCH$CHANGE_CUR_PRIORITY.

EXE$SETPRI reports a set priority system event for the target kernel thread by invoking SCH$REPORT_EVENT with a priority boost class of PRI$_IOCOM. SCH$REPORT_EVENT takes the following steps:

1. If the target kernel thread is in the COM or COMO state, SCH$REPORT_EVENT removes the kernel thread's KTB from its current COM or COMO queue. It places the KTB into the queue corresponding to the new current priority.
2. It clears and sets, as appropriate, the bits in SCH$GQ_COMQS or SCH$GQ_COMOQS.
3. It requests a rescheduling interrupt if the target kernel thread is resident and can preempt a current kernel thread.
4. If the target kernel thread is outswapped, SCH$REPORT_EVENT attempts to awaken the swapper process.

Chapter 2 further describes SCH$REPORT_EVENT's actions.

### 6.4.2 Reschedule Current Kernel Thread

The Initiate Rescheduling Interrupt ($RESCHED) system service provides run-time support for the parallel processing features of DEC FORTRAN and DEC C. It enables the currently executing kernel thread to request rescheduling, allowing other kernel threads at the same base priority to run.

The $RESCHED system service procedure, EXE$RESCHED in module SYSPARPRC, runs in kernel mode and is thread-safe. It takes the following steps:

1. It acquires the SCHED spinlock, raising IPL to IPL$_SCHED.
2. It obtains the current kernel thread's KTB.

3. It copies the contents of EXE$GL_ABSTIM_TICS, the number of 10-millisecond intervals elapsed since bootstrap, into KTB$L_ONQTIME.
4. It sets the bit KTB$V_QUANTUM_RESCHED in KTB$L_STS2.
5. EXE$RESCHED calls SCH$CHANGE_CUR_PRIORITY to lower the kernel thread's priority to its base.
6. It requests a rescheduling interrupt.
7. It releases the SCHED spinlock, restoring the previous IPL (thus enabling the rescheduling interrupt to be granted).
8. It returns a success status to its requestor.

### 6.4.3 Set Process Name

The Set Process Name ($SETPRN) system service allows a process to change or eliminate its own process name. The new name cannot contain more than 15 characters. If no other process in the same group has the same name, EXE$SETPRN, in module SYSPCNTRL, places the new name into the PCB at offset PCB$T_LNAME. Note that this service allows more flexibility in establishing a process name than is available from the usual channels, such as the user name, $JOB card, or Digital command language (DCL) command SET PROCESS /NAME, because the service imposes no restrictions on characters that can make up the process name.

The kernel threads of a process share the process name, and any of the kernel threads can change the process name. Because $SETPRN is a kernel mode system service that is not thread-safe, the kernel threads' access to the PCB is synchronized by the inner mode semaphore.

### 6.4.4 Process Mode Services

The PCB field PCB$L_STS records the current software status of the process, and the KTB field KTB$L_STS records the current software status of the kernel thread. Table 6.2 lists each of the flags in the longword and the direct or indirect ways to set or clear these flags. Each of these flags has a symbolic name of the form PCB$V_*name*, where *name* is one of those listed in the table.

The module SYSSETMOD contains three miscellaneous system services whose only action is to set or clear a bit in PCB$L_STS. These are the Set Resource Wait Mode ($SETRWM), Set System Service Failure Exception Mode ($SETSFM), and Set Swap Mode ($SETSWM) system services. Note that since these system services act only upon the PCB, they cannot be used to alter bits in KTB$L_STS for a kernel thread. To disable swapping, a process must possess the PSWAPM privilege. The other two services require no privilege.

Several system services (such as $DELPRC, $FORCEX, $RESUME, and $SUSPND) set or clear bits in PCB$L_STS as an indication that the service's primary operation has been initiated.

Table 6.2 Meanings of Flags in PCB$L_STS

| *Flag Name* | *Meaning if Set* | *Set by* | *Cleared by* |
|---|---|---|---|
| RES | Process is resident | Swapper | Swapper |
| DELPEN | Process deletion is pending | $DELPRC | |
| FORCPEN | Forced exit is pending | $FORCEX | Image rundown, Process rundown |
| INQUAN | Process is in initial quantum after inswap | Swapper | SCH$QEND |
| PSWAPM | Process swapping is disabled | $SETSWM, $CREPRC | $SETSWM |
| RESPEN | Resume is pending (skip suspend) | $RESUME | $SUSPND |
| SSFEXC | Enable system service exceptions for kernel mode | $SETSFM | $SETSFM, Process rundown |
| SSFEXCE | Enable system service exceptions for executive mode | $SETSFM | $SETSFM, Process rundown |
| SSFEXCS | Enable system service exceptions for supervisor mode | $SETSFM | $SETSFM, Process rundown |
| SSFEXCU | Enable system service exceptions for user mode | $SETSFM, $CREPRC | $SETSFM, Image rundown |
| SSRWAIT | Disable resource wait mode | $SETRWM, $CREPRC | $SETRWM |
| SUSPEN | Suspend is pending | $SUSPND | $RESUME |
| WAKEPEN | Wake is pending (skip hibernate) | $WAKE, Swapper | $HIBER Image rundown |
| WALL | Wait for all event flags in mask | $WFLAND | Next $WFLOR or $WAITFR |
| BATCH | Process is a batch job | $CREPRC | |
| NOACNT | No accounting records for this process | $CREPRC | |
| NOSUSPEND | Do not suspend this process | CWPS, Audit Server | Audit Server |
| ASTPEN | Not used | | |
| PHDRES | Process header is resident | Swapper | Swapper |
| HIBER | Hibernate after initial image activation | $CREPRC | |
| LOGIN | Log in without reading the authorization file | $CREPRC | |
| NETWRK | Process is a network job | $CREPRC | |
| PWRAST | Process has declared a power recovery AST | $SETPRA | Queuing of recovery AST, Image rundown, Process rundown |
| NODELET | Do not delete this process | CWPS, NETACP | NETACP |
| DISAWS | Disable automatic working set adjustment on this process | SET WORK /NOADJUST, $CREPRC | SET WORK /ADJUST |
| INTER | Process is interactive job | $CREPRC | |
| RECOVER | (Reserved) | | |
| SECAUDIT | Perform mandatory process auditing | LOGINOUT, $CREPRC | LOGINOUT |
| HARDAFF | Not used | | |
| ERDACT | Exec mode rundown active | Process rundown | Process rundown |
| SOFTSUSP | Process is soft suspended | $SUSPND | $RESUME, Suspend AST |
| PREEMPTED | Not used | | |

## 6.5 INTERPROCESS COMMUNICATION

In applications involving more than one process, the processes commonly share data or transfer information from one process to another. The executive provides various mechanisms that accomplish this information exchange. These mechanisms vary in the amount of information that can be transmitted, transparency of the transmission, and amount of synchronization provided by the operating system.

This section discusses event flags, lock management system services, mailboxes, logical names, and global sections. Processes running on different nodes of a VMScluster system cannot communicate using event flags, mailboxes, logical names, or global sections. The executive also provides file sharing, described in *Guide to OpenVMS File Applications*, and DECnet task-to-task communication, described in *DECnet for OpenVMS Networking Manual*.

### 6.5.1 Event Flags

Common event flags can be treated as a method for several processes to share single bits of information. A common event flag is typically used, however, as a synchronization tool for other, more complicated, communication techniques.

Common event flags can be shared by processes in the same UIC group executing on processors accessing common memory, that is, processors participating in a symmetric multiprocessing system. They cannot be shared by processes on different VMScluster nodes. Chapter 3 contains more information on the implementation of common event flags.

### 6.5.2 Lock Management System Services

Lock management system services enable a process to name an arbitrary resource and share it VMScluster-wide. A process can request locks on the named resource in a variety of lock modes to control the manner in which the process shares the resource with other processes.

A lock is owned by a process and is available to all of the process's kernel threads. Any of the process's kernel threads can acquire the lock initially, convert the lock to a different lock mode, or release the lock. When requesting a lock, a kernel thread can declare a blocking AST procedure, which is called if the granted lock blocks another request for the resource. The kernel thread can also specify the system service behavior when access to a resource cannot be granted immediately: either to wait the kernel thread until the resource is available or to return immediately with notification of the failure.

Each resource includes a 16-byte area available to store process data. The lock management system services synchronize access to this area, allowing cooperating processes with appropriate access rights to read and write the area using lock value blocks.

Chapter 4 describes the implementation of the lock management system services. Appendix A provides examples of operating system modules that use lock management system services to coordinate access to system resources.

### 6.5.3 Mailboxes

Mailboxes are software-implemented I/O devices that can be read and written through Record Management Services (RMS) requests or the Queue I/O Request (SQIO) system service on the local node. Although process-specific or systemwide parameters may control the amount of data that can be written to a mailbox in one operation, there is no limit to the total amount of information that can be passed through a mailbox with a series of reads and writes. Processes running on different nodes of a VMScluster system cannot share a mailbox.

Typically, one process reads messages written to a mailbox by one or more other processes. In the simple method of synchronizing mailbox I/O, the receiving process initiates its read of the mailbox and waits until the read completes. The read completes when another process writes to the mailbox. Since the receiving process cannot do anything else while waiting for data, this technique is restrictive.

In most applications, the receiving process performs other tasks in addition to servicing the mailbox. Putting such a process into a wait state for the mailbox prevents it from servicing any of its other tasks. In such an application, the receiving process could read the mailbox asynchronously with AST notification. Alternatively, the receiving process could queue a read attention AST request to the mailbox driver. These techniques allow a process to continue its mainline processing and to handle mailbox requests from other processes only when such work is needed.

Chapter *Mailboxes* discusses the implementation of mailboxes, and Chapter *ASTs* describes attention ASTs.

### 6.5.4 Logical Names

The executive makes extensive use of logical names to provide device independence in the I/O subsystem. Logical names can be used for many other purposes also. For example, one process can pass information to another process by creating a logical name in a shared logical name table and storing information in the equivalence string. The receiving process simply translates the name to retrieve the data.

Processes that use logical name translation as a means of interprocess communication should use proper synchronization rather than relying on an error status such as SS$_NOTRAN from the Translate Logical Name ($TRNLNM) system service for synchronization. For example, the cooperating processes can use a common event flag. An exception to this rule is

when a process creates a subprocess or detached process and passes data to the new process in the equivalence strings for SYS$INPUT, SYS$OUTPUT, or SYS$ERROR. Processes running on different nodes of a VMScluster system cannot share a logical name. Chapter *Logical Names* provides details on the implementation of logical names.

### 6.5.5 Global Sections

Global sections provide the fastest method for one process to pass information to another process on the same system. Because the processes map the data area into their address space, no movement of data takes place; the data is shared. The sharing, however, is not transparent. Each process must map the global section, and the participating processes must agree upon a synchronization technique to coordinate the reading and writing of the global section and provide notification of new data. These can be synchronized with event flags, lock management system services, or some similar mechanism.

A global section implemented on a multiprocessor system can be simultaneously accessed by multiple processes. Synchronization in such an environment requires use of MACRO-32 interlocked statements or their equivalents in other languages, or a protocol based on event flags or locks. Processes running on different nodes of a VMScluster system cannot share a global section. Chapter *Synchronization Techniques* briefly describes synchronization of shared memory.

Chapter *Memory Management System Services* describes the implementation of global sections.

## 6.6 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIB]CWPSDEF.SDL
[LIB]PSCANCTXDEF.SDL
[STARLET]PSCANDEF.SDL
[SYS]CWPS_GETJPI.MAR
[SYS]PROCESS_SCAN.MAR
[SYS]PROCESS_SCAN_CHECK.MAR
[SYS]PROCESS_SCAN_ITMLST.MAR
[SYS]SYS_AFFINITY.C
[SYS]SYSFORCEX.MAR
[SYS]SYSGETJPI.MAR
[SYS]SYSPARPRC.MAR
[SYS]SYSPCNTRL.MAR
[SYS]SYSSETMOD.MAR
[SYS]SYSSETPRI.MAR
[SYSLOA]CWPS_SERVICE_RECV.MAR

# 7 Process Creation

All things in the world come from being.
And being comes from non-being.
Lao-tzu, *Tao Tè Ching*

This chapter describes the phases in which a new process is created: creation in the context of an existing process, new process inswap, and finally, initialization in the context of the new process.

## 7.1 OVERVIEW

The creation of a new process takes place in several phases:

1. Creation begins in the context of an existing process's kernel thread that requests the Create Process ($CREPRC) system service. The $CREPRC system service performs the following steps:
   a. It makes privilege and quota checks.
   b. It allocates and initializes a process control block (PCB) for the new process; a kernel thread block (KTB) for its initial kernel thread; a job information block (JIB), unless it is creating a subprocess; and a process quota block (PQB). Explicit $CREPRC arguments and implicit parameters are taken from the context of the creator.
   c. It places those data structures into the scheduler database.
2. The initial scheduling state of the new process's initial kernel thread is computable outswapped (COMO). Thus, its execution is deferred until the swapper process selects the new process for "inswap." At that point, the swapper process performs the following steps:
   a. It allocates a balance set slot for the new process's process header (PHD) and physical memory for its initial working set.
   b. It creates a page table hierarchy for the new process.
   c. The swapper copies the template for the new process context from the executive image SHELL*x*K (the shell), where *x* is 8, 16, 32, or 64, the system's page size in kilobytes.
   d. It builds the PHD based on the values of various SYSGEN parameters.
   e. It requests that the new process be scheduled for execution.
3. The final steps of process initialization take place in the context of the new process's initial kernel thread in the routine EXE$PROCSTRT:
   a. EXE$PROCSTRT creates and initializes most of the process's permanent P1 space areas.

b. It copies information from the PQB to the PHD and various locations in P1 space.
c. It requests the image activator to activate the image specified in the $CREPRC request.
d. It calls the image at its entry point.

Note that if the image specified is LOGINOUT, many of the process's parameters may be overwritten by information in the user authorization file.

Figure 7.1 shows these phases of process creation and the context within which each phase occurs, as well as some of the events that lead to a $CREPRC request. Chapter 11 describes these events.

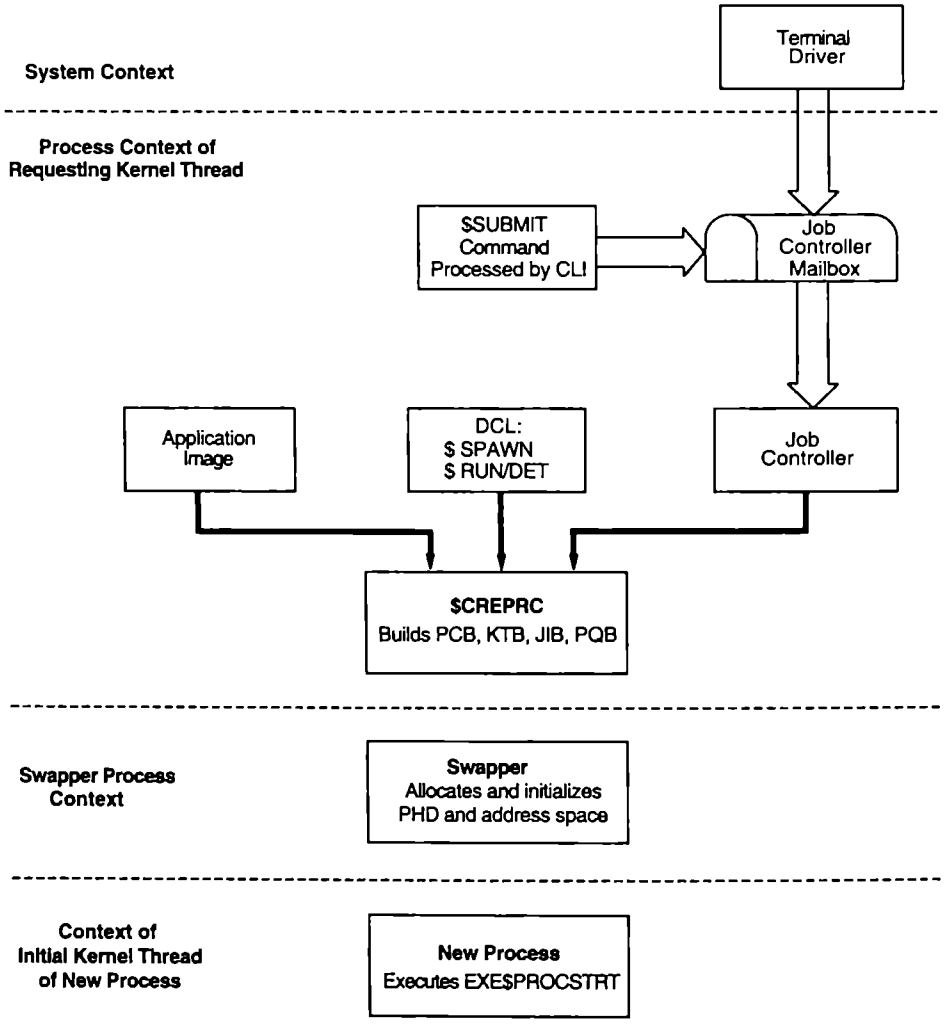



**Figure 7.1**
Process Creation

The Portable Operating System Interface (POSIX) performs a minimal process creation in response to the `fork()` POSIX system call. Support for this is implemented by EXE$CRE_MIN_PROCESS, in module SYSCREPRC, and EXE$PSX_FORK_PROCSTRT, in module POSIX_ROUTINES, in addition to some procedures described in this chapter. A description of the POSIX minimal process or its implementation is outside the scope of this book.

## 7.2 PROCESS CREATION IN THE CONTEXT OF THE CREATING PROCESS

The $CREPRC system service establishes the parameters of the new process. Some parameters are passed to the system service by the requestor. The system service copies others from the context of the requestor: the requestor's PCB, KTB, PHD, JIB, and P1 space (see Figure 7.2).

When creating a process on the local node, the $CREPRC system service can copy information to the PCB or the JIB of the new process but cannot access its PHD or P1 space because neither exists in the early stages of process creation. It stores the parameters to be copied to the PHD or P1 space in the PQB, a temporary data structure, until the new process comes into existence and has a virtual address space and a PHD. Table 7.1 lists the contents of the PQB. Its Units column specifies the unit in which each PQB quota or limit is measured.

OpenVMS Alpha Version 7.0 adds support for creating a top-level process on another VMScluster node. If the NODE argument specifies the name of another node, the $CREPRC system service stores all the parameters of the new process in a nonpaged pool data structure called a clusterwide process services data block (CWPSSRV). It uses the clusterwide process services (CWPS) mechanisms described in Chapter 6 to send the CWPSSRV to the target node.

### 7.2.1 The $CREPRC System Service

The $CREPRC system service procedure, EXE$CREPRC in module SYSCREPRC, runs in kernel mode. It performs the following steps:

1. It verifies that the address specified in the PIDADR argument is accessible to the mode from which EXE$CREPRC was requested. If not, it returns the error status SS$_ACCVIO.
2. It creates either a top-level process, detached from its creator, or a subprocess, attached to its creator's job tree. EXE$CREPRC's actions depend on the UIC argument and the PRC$V_DETACH bit in the STSFLG argument.
   - If the requestor specified a nonzero UIC argument, EXE$CREPRC creates a top-level process. The process is further classified as interactive, network, batch, or detached based on EXE$CREPRC's STSFLG argument.

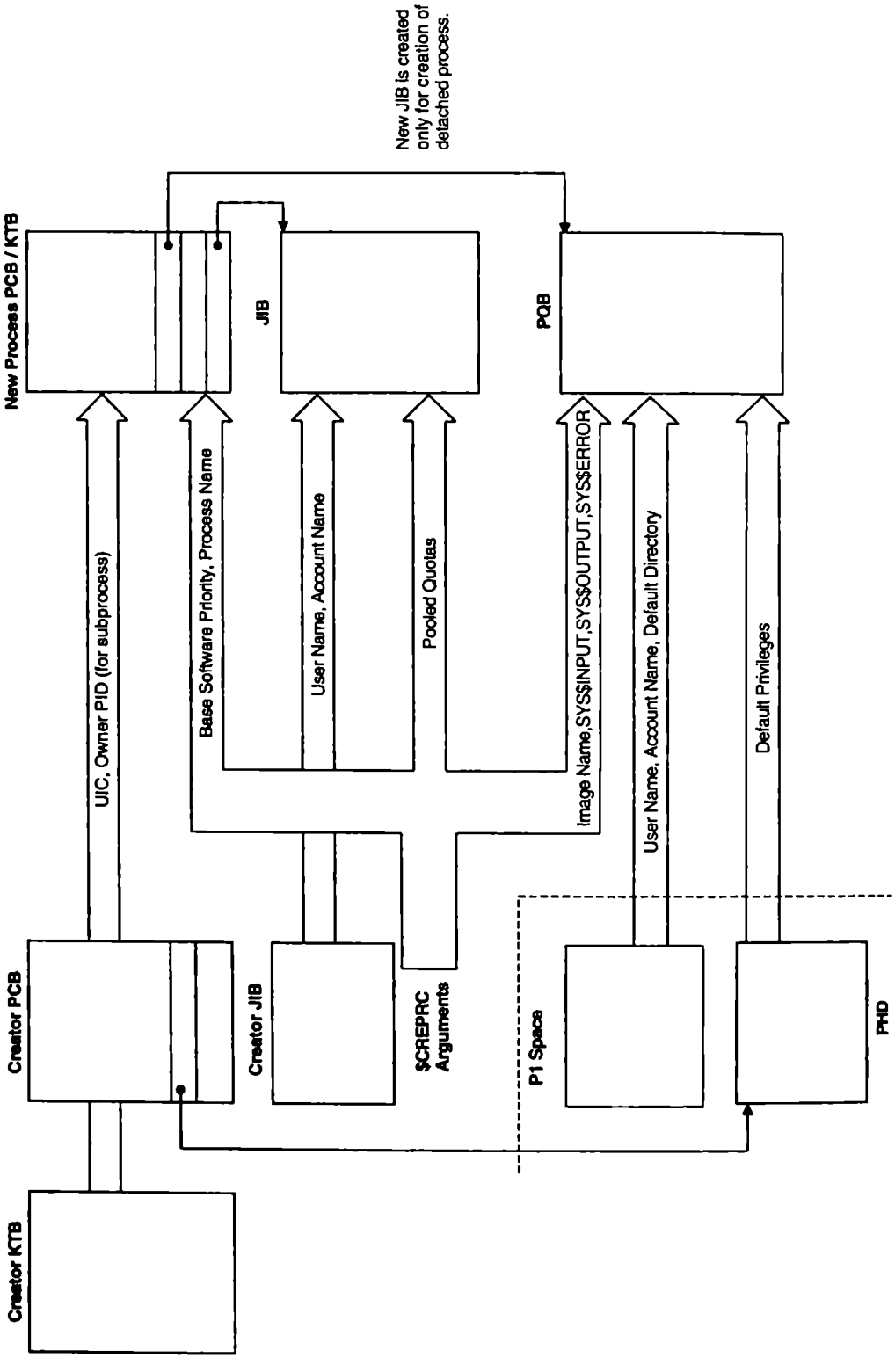

New JIB is created only for creation of detached process.
New Process PCB / KTB
JIB
PQB
UIC, Owner PID (for subprocess)
Base Software Priority, Process Name
User Name, Account Name
Pooled Quotas
Image Name,SYS$INPUT,SYS$OUTPUT,SYS$ERROR
User Name, Account Name, Default Directory
Default Privileges
Creator PCB
Creator JIB
$CREPRC Arguments
P1 Space
PHD
Creator KTB

Table 7.1 Contents of the Process Quota Block

| *Item* | *Size (bytes)* | *Units* |
|---|---|---|
| Privilege mask | 8 | — |
| Size of PQB | 2 | Bytes |
| Type code | 1 | — |
| Status | 1 | — |
| AST limit | 4 | ASTs |
| Buffered I/O limit | 4 | Requests |
| Reserved | 4 | — |
| CPU time limit | 4 | 10-ms intervals (soft ticks) |
| Direct I/O limit | 4 | Requests |
| Reserved | 16 | — |
| Working set quota | 4 | Pagelets |
| Working set default | 4 | Pagelets |
| Lock limit | 4 | Locks |
| Working set extent | 4 | Pagelets |
| Logical name table quota | 4 | Bytes |
| Flags | 2 | — |
| Default message flags | 1 | — |
| Reserved | 1 | — |
| Authorization file flags | 4 | — |
| Process creation flags | 4 | — |
| Minimum authorized security class | 20 | — |
| Maximum authorized security class | 20 | — |
| SYS$INPUT attributes | 4 | — |
| SYS$OUTPUT attributes | 4 | — |
| SYS$ERROR attributes | 4 | — |
| SYS$DISK attributes | 4 | — |
| CLI image name | 32 | — |
| CLI command table name | 256 | — |
| Spawn CLI image name | 32 | — |
| Spawn CLI command table name | 256 | — |
| Equivalence name for SYS$INPUT | 256 | — |
| Equivalence name for SYS$OUTPUT | 256 | — |
| Equivalence name for SYS$ERROR | 256 | — |
| Equivalence name for SYS$DISK | 256 | — |
| Default directory string | 256 | — |
| Image name | 256 | — |
| Account name | 8 | — |

- If the UIC argument is zero, the default, and the requestor did not set the PRC$V_DETACH bit in the STSFLG argument, EXE$CREPRC creates a subprocess.
- If the UIC argument is zero, the default, but the requestor set the PRC$V_DETACH bit in the STSFLG argument, EXE$CREPRC creates a top-level, detached process with the same user identification code (UIC) as that of the requestor.

EXE$CREPRC tests whether the specified UIC is zero or the same as that of the requestor. If it is, no privilege is necessary to create a top-level process. Otherwise, the requestor needs either the DETACH or CMKRNL privilege. If the requestor requested creation of a top-level process without the necessary privilege, EXE$CREPRC returns the error status SS$_NOPRIV.

3. It tests if any undefined STSFLG argument bits have been set and, if so, returns the error status SS$_IVSTSFLG.
4. It copies the status flags for the new process from the STSFLG argument to local storage and confirms that the process has any privileges required to specify particular flags, returning SS$_NOPRIV if not.
5. It determines the process privileges of the new process and stores them in local storage.

   - If no privilege argument is present, EXE$CREPRC uses the current working privileges: the combination of image privileges and those of the current process.
   - If a privilege argument is present and the creator has SETPRV privilege, EXE$CREPRC uses the privilege argument with no modification.
   - If a privilege argument is present and the creator does not have SETPRV privilege, EXE$CREPRC stores the logical AND of the working privileges of the creator and the privileges specified in the argument. When the creator lacks SETPRV, a process cannot receive privileges that its creator does not have.

   Table 8.2 summarizes the various privilege masks associated with a process.
6. EXE$CREPRC determines the software priority of the new process and saves it in local storage. The system service macro $CREPRC_S, used from MACRO-32, specifies a default value of 2 for the BASPRI argument if it was omitted. The default value for other languages is determined by the treatment of missing arguments by the language processor. Typically, the default value is 0.

   OpenVMS components that create processes typically supply an explicit priority. For example, the job controller specifies the value of the DEFPRI SYSGEN parameter.

   If the creator has ALTPRI privilege, EXE$CREPRC uses the priority specified in the argument list. If the creator does not have ALTPRI privilege, EXE$CREPRC uses the smaller of the creator's base priority and the priority in the argument list.
7. It allocates a kernel request packet (KRP), translates the logical name SYS$DISK in the table LNM$FILE_DEV, and saves its equivalence name in the KRP. For compatibility with earlier OpenVMS releases, SYS$DISK is translated only once. Thus, its equivalence name must be either a shareable logical name or a physical device name.

8. It confirms that the PRCNAM argument, if present, has 15 or fewer characters, returning SS$_IVLOGNAM if not.

   If the argument is present, it confirms that the name string is accessible to the service requestor, returning SS$_ACCVIO if not.
9. It determines whether this is a request to create a process on the local node or on another VMScluster node, based on the presence and value of the NODE argument.

   - If the argument is present, it confirms that the name string is accessible to the service requestor, returning SS$_ACCVIO if not.
   - It confirms that the NODE argument, if present, has six or fewer characters, returning SS$_IVLOGNAM if not.
   - If the argument is present, but it specifies a nonexistent node or if the local node is not a VMScluster member, EXE$CREPRC returns SS$_NOSUCHNODE.
   - If the argument specifies another node, EXE$CREPRC allocates a CWPSSRV from nonpaged pool and initializes it. It copies the service arguments and process parameters from the current process's context to the CWPSSRV and sends it to the target node, using the CWPS mechanism. The requesting kernel thread is placed into a resource wait state (MWAIT) for resource RSN$_CLUSRV.

     A cooperating CWPS routine on the other node processes the request. The details of its processing are beyond the scope of this book. The routine transmits status and the extended process ID (EPID) of the created process back to this node. Through mechanisms described in Chapter 6, the waiting kernel thread is made computable, and control returns to EXE$CREPRC, which returns the status and EPID to its requestor.
   - If the NODE argument is absent or specifies the local node, EXE$CREPRC continues with the next steps to create a process on this node.

10. EXE$CREPRC raises interrupt priority level (IPL) to 2 and allocates a PQB from either the PQB lookaside list or paged pool. It copies the translation of SYS$DISK to the PQB and deallocates the KRP.

    Next, it allocates a PCB from nonpaged pool. The PCB also serves as the KTB for the initial kernel thread in the new process. EXE$CREPRC remains at IPL 2 or above from this point to prevent process deletion and the loss of allocated but unrecorded memory. Subsequently, if an error occurs, EXE$CREPRC deallocates the PCB, PQB, and JIB (if one was allocated) before returning the error status to the requestor.

    Chapter *Pool Management* describes nonpaged pool, paged pool, the PQB lookaside list, and KRPs.
11. EXE$CREPRC stores the priority of the new process's initial kernel thread in KTB$L_PRIB and KTB$L_PRI.

Table 7.2 Status Flags Specified at Process Creation

| *Flag Argument* | *Meaning if Set* | *Destination* |
|---|---|---|
| PRC$V_SSRWAIT | Disable system service resource wait mode | PCB$V_SSRWAIT |
| PRC$V_SSFEXCU | Enable system service exceptions for user mode | PCB$V_SSFEXCU |
| PRC$V_PSWAPM[1] | Inhibit process swapping | PCB$V_PSWAPM |
| PRC$V_NOACNT[2] | Suppress accounting | PCB$V_NOACNT |
| PRC$V_BATCH[3] | Batch (noninteractive) process | PCB$V_BATCH |
| PRC$V_HIBER | Hibernate process before calling image | PCB$V_HIBER |
| PRC$V_DEBUG | RUN requested with either /DEBUG or /NODEBUG | Passed to $IMGSTA |
| PRC$V_DBGTRU | RUN qualifier was /DEBUG | Passed to $IMGSTA |
| PRC$V_NOUAF | Log in without reading the authorization file | PCB$V_LOGIN |
| PRC$V_NETWRK[3] | Process is associated with an incoming network connection | PCB$V_NETWRK |
| PRC$V_DISAWS | Disable system-initiated working set list adjustment | PCB$V_DISAWS |
| PRC$V_DETACH[4] | Process is detached | PCB$V_DETACH |
| PRC$V_INTER | Process is interactive | PCB$V_INTER |
| PRC$V_CLISPEC | Copy CLI command table name for use by subprocesses | |
| PRC$V_IMGDMP | Enable image dump | PHD$V_IMGDMP |
| PRC$V_NOPASSWORD | Disable prompt for user name and password | PCB$V_NOPASSWORD |
| PRC$V_SUBSYSTEM | Inherit subsystem identifiers | |
| PRC$V_TCB[3] | Process is part of the OpenVMS trusted computing base | PCB$V_TCB[5] |

[1] Requires PSWAPM privilege.
[2] Requires NOACNT privilege.
[3] Requires DETACH privilege.
[4] Flag ignored unless same UIC.
[5] Flag is in PCB$L_STS2.

12. It extracts the status flags for the new process from local storage and sets the corresponding flags in the PCB. It always propagates the flag PCB$V_SECAUDIT from the creator process. Table 7.2 describes the status flags. Most flags listed in the table are stored in the field PCB$L_STS. Several, however, are temporarily stored in the PQB because their ultimate destination is in process-private address space that does not yet exist.
13. It copies the access rights block (ARB) from the creating process's ARB. If the creator has an extended rights list, EXE$CREPRC allocates a nonpaged pool buffer into which it copies the extended rights list. It initializes the new process's ARB fields that describe the extended rights list.

If the current process is running an image that is a trusted subsystem (its image file has a subsystem access control entry) and if the PRC$V_SUBSYSTEM flag was set in the STSFLG argument, EXE$CREPRC allocates a nonpaged pool buffer into which it copies the image rights. It initializes the ARB fields that describe the image rights list.

The ARB is currently located within the PCB. However, executive routines that check a process's access rights use the ARB pointer, PCB$L_ARB, to locate the process rights and UIC. All programs should follow this convention, since the ARB may become an independent structure in the future. Any programs that do not use the ARB pointer would require modification if this occurs.

14. EXE$CREPRC determines the UIC of the new process and stores it in the PCB. The UIC argument is used if the requestor specified that argument. Otherwise, EXE$CREPRC uses the UIC of the creator. Therefore, a subprocess always has the same UIC as its creator—if the UIC argument had been specified, EXE$CREPRC would have created a top-level process.
15. If the new process is a subprocess, EXE$CREPRC copies the internal process ID (IPID) and the EPID of the creator process (namely, that of its initial kernel thread) to the PCB$L_OWNER and PCB$L_EOWNER fields of the new PCB. Section 7.3 describes internal and extended process IDs.

    If the new process is a top-level process, the PCB$L_OWNER and PCB$L_EOWNER fields remain zero.
16. JIB initialization for top-level processes differs from that for subprocesses:

    - If EXE$CREPRC is creating a top-level process, it allocates a JIB from nonpaged pool. It initializes the JIB's jobwide list of mounted volumes as an empty list, then copies the account and user name fields from the creating process's JIB and zeros the rest of the JIB.
    - If EXE$CREPRC is creating a subprocess, no JIB allocation is necessary; the subprocess shares its creator's JIB. EXE$CREPRC increments JIB$L_PRCCNT, the count of subprocesses in the job tree.

    Figure 7.3 shows the layout of the JIB, and Figure 7.4 shows the relation between the JIB and the PCBs of several processes in the same job.
17. If the new process is a subprocess, EXE$CREPRC propagates the state of the shelving bits, PCB$V_NOUNSHELVE and PCB$V_SHELVING_RESERVE, to it.

    These bits are intended for use by the layered product Polycenter HSM for OpenVMS. This product provides hierarchical storage management features, which enable inactive files to be archived to tape and retrieved when accessed. PCB$V_NOUNSHELVE, when set, means that when an archived file is accessed, it should not automatically be restored to disk. The other bit is reserved.

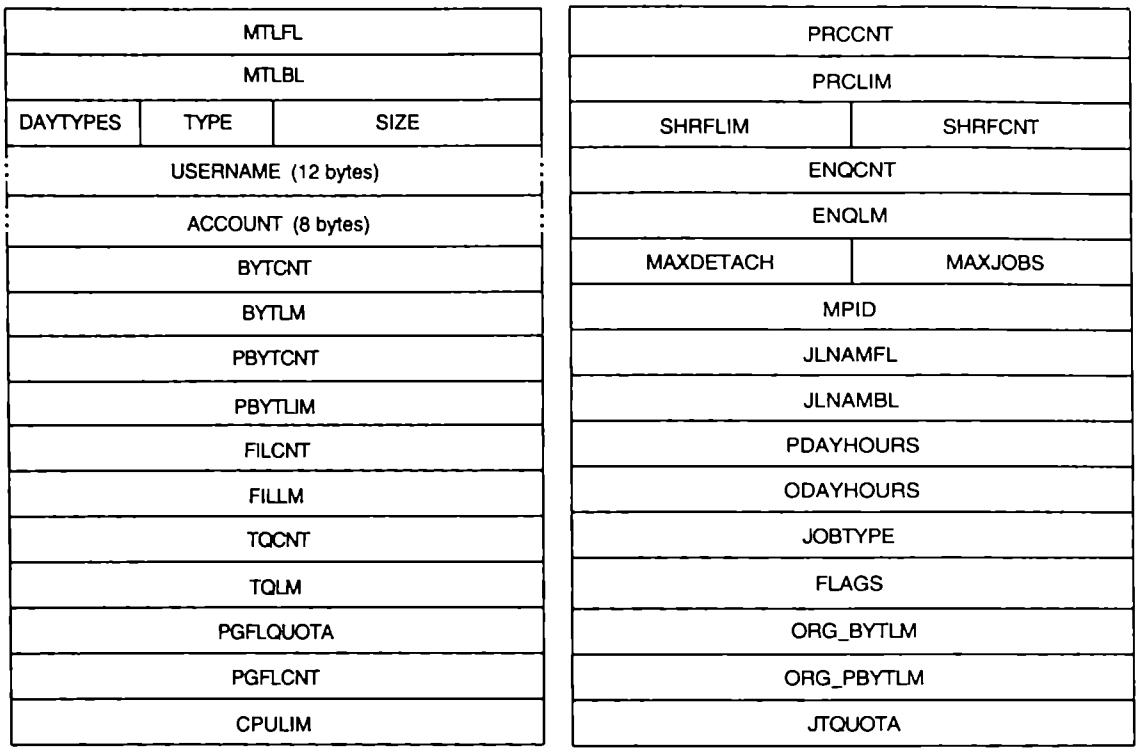



**Figure 7.3**
Layout of a Job Information Block

18. If creating a subprocess, EXE$CREPRC charges JIB$L_PGFLCNT, the job page file quota, for the number of process page file pages contributed by the shell.

    If the job has insufficient page file quota, EXE$CREPRC deallocates the newly acquired data structures and returns the error status SS$_EXQUOTA to its requestor. Otherwise, it increments JIB$L_PRCCNT and compares it to JIB$L_PRCLIM, the maximum number of subprocesses in the job tree. If JIB$L_PRCCNT exceeds JIB$L_PRCLIM, the job tree is at its maximum size. EXE$CREPRC decrements JIB$L_PRCCNT, deallocates the PCB and PQB, returns page file quota, and returns the error status SS$_EXQUOTA to its requestor.

    Note that the process count field within a PCB (PCB$L_PRCCNT) tracks the number of subprocesses created by one process. JIB$L_PRCCNT counts the total number of subprocesses in the entire job.
19. For both top-level processes and subprocesses, EXE$CREPRC stores the address of the JIB in PCB$L_JIB.
20. It copies the default file protection from the creating process's PCB to the new process's PCB.

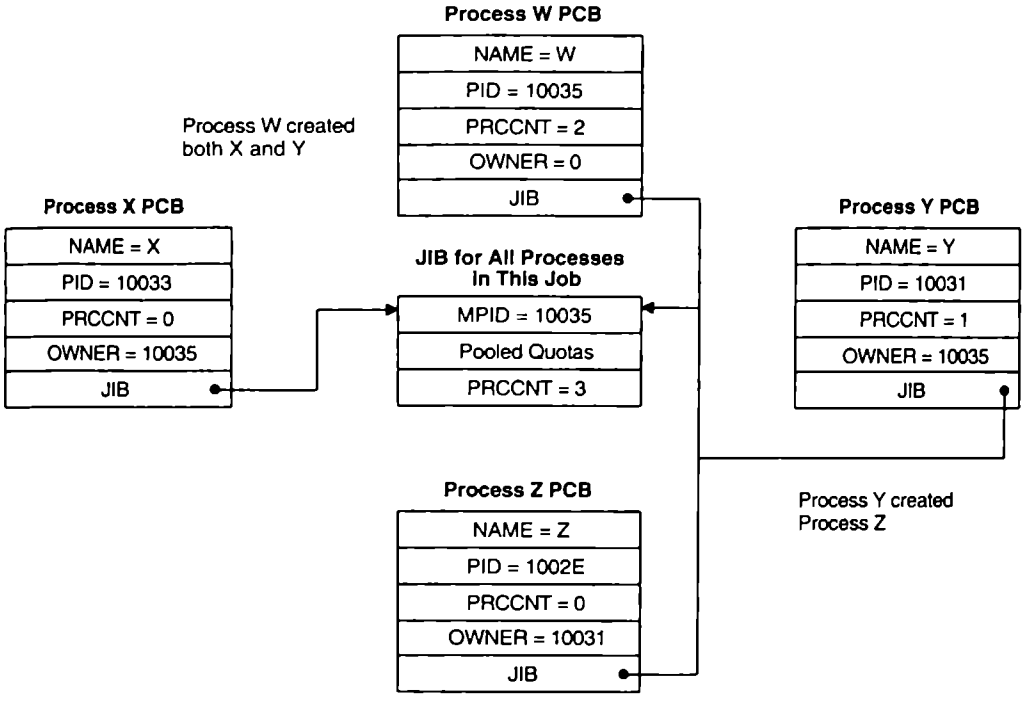



**Figure 7.4**
Relation Between the JIB and the PCBs of Several Processes in the Same Job

21. EXE$CREPRC copies the creator's account name from CTL$T_ACCOUNT to the PQB.
22. EXE$CREPRC copies the minimum and maximum authorized security clearance records from the creator's PHD to the new process's PQB.
23. It copies the following information from the P1 space of the creator process:
    - Default directory string
    - Command language interpreter (CLI) name
    - Command table name
    - CLI name for use by spawned subprocesses
    - Command table name for use by spawned subprocesses
24. It copies the default message flags and flags specified in the creator's authorization file record from the P1 space of the creator to the PQB.
25. It copies the saved STSFLG argument to PQB$L_CREPRC_FLAGS. It propagates the values of PRC$V_IMGDMP, PRC$V_DEBUG, and PRC$V_DBGTRU from the STSFLG argument to PQB$W_FLAGS.
26. It records the privileges of the new process in the PQB.

27. EXE$CREPRC copies the unit number of the termination mailbox from the MBXUNIT argument. The termination mailbox is used when the process is eventually deleted. At that time, the process deletion routine writes a termination message to the specified mailbox if the unit number is nonzero.
28. EXE$CREPRC copies several text strings to the PQB, taking the image name and the equivalence names for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR from the $CREPRC argument list. For most processes, the image is LOGINOUT.EXE.
29. If the process being created is not a subprocess, and it is not a batch, network, or interactive process, then it must be a true detached process. In that case, EXE$CREPRC copies JIB$W_MAXJOBS and JIB$W_MAXDETACH from the creator's JIB to that of the new process. If either count is nonzero, indicating a limit, EXE$CREPRC must check whether creation of this process would exceed one of those limits.

    It acquires the SCHED spinlock, raising IPL to IPL$_SCHED. Holding the spinlock, it scans all existing processes except the swapper process. It looks for a process that is not a network process or a subprocess and that has the same user name as the process being created. If it finds one, it increments the total count of jobs with that user name. If the process is neither interactive nor batch, it also increments the total count of detached processes with that user name.

    After scanning all the processes, EXE$CREPRC releases the SCHED spinlock and restores IPL to 2. If either job limit has been exceeded, EXE$CREPRC deallocates the structures it allocated and returns the error status SS$_EXPRCLM to its requestor.
30. It determines the quotas for the new process and stores them in the PQB. Section 7.2.2 describes the steps taken to determine the quota list for the new process.
31. EXE$CREPRC processes the ITMLST argument, if one was supplied. This argument is reserved for use by the executive to pass logical name attributes for SYS$INPUT, SYS$OUTPUT, and SYS$ERROR to EXE$CREPRC, which in turn copies the attributes into the PQB.
32. It explicitly initializes several fields in the PCB and the initial kernel thread's KTB:

    a. It sets up five asynchronous system trap (AST) queues, one for each of the four access modes and one for special kernel mode ASTs, as empty listheads in the KTB.
    b. It also initializes empty listheads in the PCB for processwide special kernel mode, kernel mode, and executive mode ASTs (see Chapter 5).
    c. It sets up the PCB's lock queue as an empty listhead.

d. It sets up the PCB's buffer object queue (see Chapter *Memory Management Data Structures*) as an empty listhead.
e. EXE$CREPRC initializes the PCB's resource domain pointer block (RDPB) queue as an empty listhead. A resource domain is a name space for root resource names (see Chapter 4).
f. It initializes the PCB fields that record the number of kernel threads in the process (PCB$L_KT_COUNT), highest ID of a kernel thread (PCB$L_KT_HIGH), address of the array pointing to KTBs of the process (PCB$L_KTBVEC), and the address of the initial kernel thread KTB (PCB$L_INITIAL_KTB).

   It stores the address of the permanent per-kernel thread area in KTB$L_PER_KT_AREA.

   Chapter 5 contains more information on support for kernel threads.
g. It stores a −1 in PCB$Q_KEEP_IN_WS and PCB$Q_KEEP_IN_WS2 to indicate no current address range to be retained in the process's working set.
h. It initializes the process-private page count, PCB$L_PPGCNT, to the number of pages required for the new process header and the shell pages.
i. It clears the initial kernel thread's current and permanent CPU affinities.

   It initializes the initial kernel thread's current and permanent CPU capability requirement fields to the system default value found in SCH$GL_DEFAULT_PROCESS_CAP. It clears the saved capabilities.

   It copies the default affinity skip value from SCH$GL_AFFINITY_SKIP to the KTB.

   If the system default capability mask enables implicit affinity, it calls SCH$SELECT_IMPLICIT_CPU, in module CAPABILITY, to select a CPU on which the kernel thread can run and stores its ID in KTB$L_BIAS_CELL. If implicit affinity is not enabled, it initializes the field to −1 to indicate implicit affinity is disabled.

   Chapter 2 describes process affinity and capabilities.
j. If the process being created is part of the OpenVMS trusted computing base (if PCB$V_TCB in PCB$L_STS2 is set), it initializes PCB$L_NOAUDIT to 1 to disable auditing for the process.
k. EXE$CREPRC copies the process name, if one was specified in the PRCNAM argument, to PCB$T_LNAME.

33. EXE$CREPRC tests that the process name is unique within the UIC group. It examines the process name fields of all PCBs in the system with the same group number. If the process name is not unique,

EXE$CREPRC deallocates the structures it allocated and returns the error status SS$_DUPLNAM to its requestor. Process name is always qualified by UIC group number.

34. It stores the address of the PQB in the field PCB$L_PQB. PCB$L_PQB is the same longword as PCB$L_FREWSLE_PARAM, a field that is not used before image execution.
35. EXE$CREPRC acquires the MMG and SCHED spinlocks, raising IPL to IPL$_MMG. It searches the PCB vector for an empty slot (see Section 7.3.1). If none is available, it deallocates the structures it allocated and returns the error status SS$_NOSLOT to its requestor after releasing the SCHED and MMG spinlocks. The PCB vector is pictured in Figure 2.1.

    Otherwise, having found an available PCB vector slot, EXE$CREPRC tests the maximum process count. If the maximum process count has been exceeded (if SCH$GW_PROCCNT's contents are larger than those of SCH$GW_PROCLIM), EXE$CREPRC deallocates the structures it allocated and returns the error status SS$_NOSLOT to its requestor after releasing the SCHED and MMG spinlocks.

    If the count has not been exceeded, EXE$CREPRC increments SCH$GW_PROCCNT.

    If the new process is interactive, EXE$CREPRC increments SYS$GL_IJOBCNT, the current interactive job count for the system. Since all interactive jobs begin by executing the LOGINOUT image, the comparison of SYS$GL_IJOBCNT to the SYSGEN parameter IJOBLIM is handled by LOGINOUT.

    If the new process is a batch job, EXE$CREPRC increments SYS$GL_BJOBCNT, the current batch job count for the system.
36. EXE$CREPRC stores the new PCB address in the available PCB vector slot.
37. It fabricates internal and extended process IDs (see Section 7.3.1) and stores them in the PCB of the new process.
38. If the new process is not a subprocess, EXE$CREPRC stores its IPID in the master process ID field of the JIB (JIB$L_MPID).
39. EXE$CREPRC calls routine SCH$CHSE, in module RSE, to insert the new KTB into the COMO scheduling queue. It specifies the priority increment class PRI$_TICOM to give a boost of 6 to the base priority (see Chapter 2).
40. If it is creating a subprocess, EXE$CREPRC increments the count of subprocesses owned by the creator (PCB$L_PRCCNT in the creator's PCB). In addition, if a CPU time limit is in effect for the creator, EXE$CREPRC deducts the amount of CPU time passed to the new process from the creator.
41. Finally, it releases the SCHED and MMG spinlocks and returns the EPID of the new process (if requested) to the requestor.

### 7.2.2 Establishing Quotas for the New Process

The $CREPRC system service uses two tables in the executive to set up quotas for a new process: a minimum quota table and a default quota table. Each quota or limit has an entry in both tables. The contents of the minimum table are determined by the SYSGEN parameters whose names are of the form PQL_M*quota-name*; the contents of the default table are of the form PQL_D*quota-name*.

A quota can be pooled, deductible, or neither. A pooled quota is shared among all processes in a job. A deductible quota is charged against a process when it creates a subprocess.

Following is a list of the steps EXE$CREPRC takes to determine the value for each quota or limit that is passed to a process created on the local node:

1. It places the default value for each quota into the PQB as an initial value.
2. It replaces the default values in the PQB by any quotas specified in the QUOTA argument list to the $CREPRC system service.
3. It forces each quota to at least its minimum value.
4. It checks to ensure that the creator possesses sufficient quota to cover the quota that it is giving to the new process:
   - If the creator has either DETACH or CMKRNL privilege and is creating a top-level process, quotas are unrestricted and no check is performed.
   - If the creator has neither privilege and is creating a top-level process, then the new process quotas must be less than or equal to those of the creator.
   - If a subprocess is being created and the quota is neither pooled nor deductible (the only deductible quota currently implemented is CPU time limit), then the subprocess quota must be less than or equal to the creator's quota.
   - Pooled quotas require no special action when a subprocess is being created because they already reside in the JIB, a structure that is shared by all processes in the job (see Figure 7.4).
   - If a subprocess is being created and the quota in question is the CPU time limit quota, EXE$CREPRC's actions depend on how much quota the creator process possesses. If the creator has an infinite CPU time limit, then no check is performed. If the creator has a finite CPU time limit and specifies an infinite CPU time limit for the subprocess, half of the creator's remaining CPU time is passed to the subprocess. If the creator has a finite CPU time limit and specifies a finite CPU time limit for the subprocess, the amount passed to the subprocess must be less than the creator's original quota, or the creation is aborted with the error SS$_EXQUOTA.
5. EXE$CREPRC places pooled quotas directly into the JIB, if newly allocated. It places other quotas into the PCB or stores them temporarily in the PQB.

Table 7.3 Storage Areas for Process Quotas

| *Quota/Limit Name* | *Location of Active Count* | *Count Stored by*[1] | *Location of Process Limit* | *Limit Stored by*[1] |
|---|---|---|---|---|
| | | *Nondeductible Quotas* | | |
| AST limit | PCB$L_ASTCNT | C | PHD$L_ASTLM | P |
| Buffered I/O limit | PCB$L_BIOCNT | C | PCB$L_BIOLM | C |
| Direct I/O limit | PCB$L_DIOCNT | C | PCB$L_DIOLM | C |
| Working set quota | n/a[2] | – | PHD$L_WSQUOTA | P |
| Working set default | n/a[2] | – | PHD$L_DFWSCNT | P |
| Working set extent | n/a[2] | – | PHD$L_WSEXTENT | P |
| | | *Deductible Quota* | | |
| CPU time limit[3] | PHD$L_CPUTIM | P | PHD$L_CPULIM | P |
| | | *Pooled Quotas* | | |
| Buffered I/O byte limit | JIB$L_BYTCNT | C[4] | JIB$L_BYTLM | C[4] |
| Open file limit | JIB$L_FILCNT | C[4] | JIB$L_FILLM | C[4] |
| Page file page limit | JIB$L_PGFLCNT | C[4] | JIB$L_PGFLQUOTA | C[4] |
| Subprocess limit | JIB$L_PRCCNT | C[4] | JIB$L_PRCLIM | C[4] |
| Timer queue entry limit | JIB$L_TQCNT | C[4] | JIB$L_TQLM | C[4] |
| Enqueue limit | JIB$L_ENQCNT | C[4] | JIB$L_ENQLM | C[4] |

[1] C indicates that the value is stored by EXE$CREPRC. P indicates that the value is stored by EXE$PROCSTRT.

[2] Working set list quotas are handled differently from other quotas (see Chapter *Working Set List Dynamics*).

[3] CPUTIM starts at zero and increments for each 10-ms (soft tick) interval that the process is current. If limit checking is in effect (CPULIM nonzero), then CPUTIM may not exceed CPULIM.

[4] The contents of the JIB are loaded by EXE$CREPRC when a detached process is created. Subprocess creation uses an existing JIB.

Table 7.3 lists the quotas that are passed to a new process when it is created, whether each quota is deductible or pooled, and where the limit is stored in the context of the new process. Further discussion of quotas can be found in the *OpenVMS System Manager's Manual* and in the *OpenVMS System Services Reference Manual*.

With the exception of CPU time limit and subprocess count, all active counts start at their process limit values and decrement to zero. An active count of zero indicates no quota remaining. An active count equal to the corresponding process limit indicates no resources associated with the quota are in use.

In creating a process on another VMScluster node, EXE$CREPRC takes the following steps to check quotas:

1. It stores any quotas specified by the $CREPRC argument list in the CWPSSRV.

2. It checks that the creator is allowed to specify as much of each quota as it did, minimizing any quota as necessary:
   - If the creator has either DETACH or CMKRNL privilege, quotas are unrestricted and no check is performed.
   - If the creator has neither privilege, the new process quotas must be less than or equal to those of the creator.
3. On the remote node, the quotas passed in the CWPSSRV are maximized with the default quota values from that node's SYSGEN parameters, thus allowing for differences based on different CPU configurations. Any missing values are supplied from the remote node's default quota values.
4. On the remote node, each resulting quota is forced to at least its minimum value based on that node's SYSGEN parameters.
5. On the remote node, quota values are stored in the appropriate process data structures.

## 7.3 PROCESS IDENTIFICATION

The executive provides two forms of process identifier (PID) for each process. The internal, traditional form—the IPID—identifies a process or kernel thread within the context of a single system. The EPID is a compressed version of the IPID that additionally identifies the VMScluster node of a process or kernel thread. In this book, the unqualified term *process ID* or *PID* refers to the internal, traditional form.

Each PCB contains four fields related to process identification, all initialized by EXE$CREPRC:

- PCB$L_PID—Internal process ID
- PCB$L_EPID—Extended process ID
- PCB$L_OWNER—Internal process ID of process's creator
- PCB$L_EOWNER—Extended process ID of process's creator

The executive generally identifies a process internally by its IPID, although code such as the lock management and clusterwide process system services may use both forms of PID. System services accept and return EPIDs, and system utilities display EPIDs, but the format of the EPID is subject to change in future versions of the operating system. No program should attempt to interpret individual EPID fields.

The executive provides the following routines (in the module SYSPCNTRL) for transformation or manipulation of EPIDs and IPIDs by kernel mode code:

- EXE[_STD]$CVT_EPID_TO_IPID—Convert an EPID to IPID
- EXE[_STD]$CVT_EPID_TO_KTB—Convert an EPID to address of corresponding KTB

- EXE[_STD]$CVT_EPID_TO_PCB—Convert an EPID to address of corresponding PCB
- EXE[_STD]$CVT_IPID_TO_EPID—Convert an IPID to EPID
- EXE[_STD]$CVT_IPID_TO_KTB—Convert an IPID to address of corresponding KTB
- EXE[_STD]$CVT_IPID_TO_PCB—Convert an IPID to address of corresponding PCB

### 7.3.1 Internal PID

All process PCB addresses are stored in an array called the PCB vector (see Figure 2.1). The number of entries in that array and a related array called the sequence vector (and therefore the maximum number of processes allowed at any given time on a system) is determined by the SYSGEN parameter MAXPROCESSCNT.

When EXE$CREPRC creates a process, it searches the PCB vector for an unused entry in which to insert the address of the new PCB it has built. EXE$CREPRC excludes the first two PCBs (the null PCB and the swapper PCB) from its scan of the PCB vector. It begins the scan with the entry most recently allocated and wraps to the entry after the swapper process if it exceeds the maximum entry. The index of the maximum entry is stored in SCH$GL_MAXPIX.

The index of the first free entry it finds becomes the low-order word of the new process's IPID, the process index. The high-order word comes from the corresponding sequence number vector. Figure 7.5 shows how an IPID is constructed.

As processes are created and deleted on the system over time, the entries in the PCB vector are reused. The sequence vector tracks the reuse of these PCB vector entries. In previous OpenVMS versions, a process had one sequence number associated with its use of the process index. Each time a process was created and assigned to a PCB vector entry, the sequence number was incremented. In OpenVMS Alpha Version 7.0, a process can have a range of sequence numbers associated with its use of the process index, one sequence number for each kernel thread in the process. This enables each kernel thread to have a unique IPID. The IPID of the initial kernel thread is the same as the process's IPID.

Each entry in the sequence vector is initialized during system initialization: the low-order word, the base sequence number field, is zeroed; the high-order word, the next sequence number to use, is set to 1. At the first use of any process index, EXE$CREPRC increments the base sequence number field to 1, stores that value in the high-order word of the IPID, and increments the next sequence number field's contents to 2. On subsequent uses of a process index, EXE$CREPRC simply copies the base sequence number to the high-order word of the IPID.

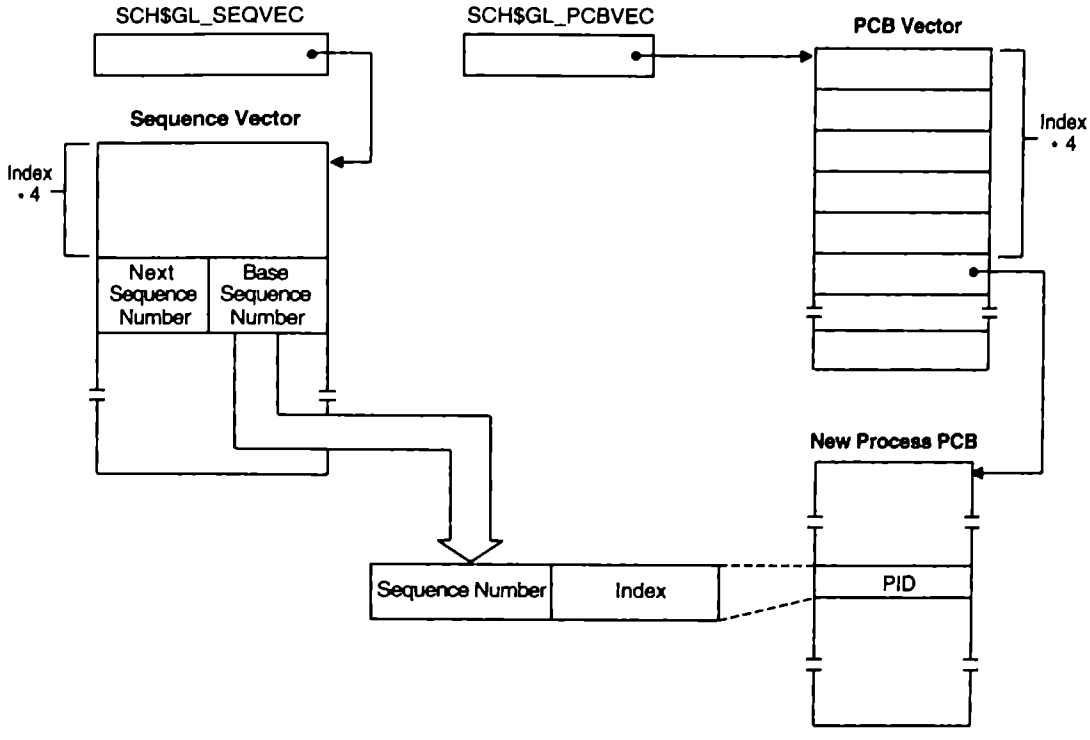



**Figure 7.5**
Fabrication of an Internal Process ID (IPID)

When a multithreaded process creates a kernel thread, the executive assigns it an available thread ID (a number from 1 to a value 1 less than the SYSGEN parameter MULTITHREAD). The executive forms the kernel thread's IPID by adding its thread ID to the sequence number in the initial kernel thread's IPID. If the resulting sequence number is equal to or greater than the next sequence number for that process index, the next sequence number is replaced by a value 1 larger than the new kernel thread's sequence number.

Within a multithreaded process, IPIDs can be reused as kernel threads are created and deleted. In particular, if a process reruns a multithreaded application, the IPIDs of the created kernel threads are the same from run to run.

When a process is deleted, the executive stores the address of the null PCB in its PCB vector slot to indicate that the slot is available. It updates the corresponding sequence number array entry by copying the next sequence number field to the base sequence number field and incrementing the next sequence number field. If the new sequence number field contains a number larger than the maximum allowed, the executive resets it to the value 1 and resets the next sequence number field to the value 2. Although the sequence number wraps when it reaches a maximum, it is believed unlikely

that any traces of the previous process assigned to an IPID could persist in the system to a time when that same IPID could be assigned to a different process.

The maximum allowed sequence number is the contents of SCH$GW_MAXPIXSEQ, 16,383 in OpenVMS Alpha Version 7.0. (Its maximum is constrained to allow for a larger process index field in the EPID.) Therefore, when an IPID is interpreted as a signed integer, it is always positive. This allows the I/O subsystem to treat a negative value in the IRP$L_PID field of an I/O request packet (IRP) in a special manner. The I/O postprocessing interrupt service routine interprets a negative IRP$L_PID value as the (32-bit system virtual) address of an internal I/O completion routine.

Executive routines use the process index part of an IPID or EPID to locate the PCB associated with a kernel thread. They use the sequence number to check the validity of an IPID. If the IPID represents a process rather than a kernel thread, the high-order word of the IPID must match the base sequence number field in the sequence number vector entry indexed by the low-order word of the IPID. Additionally, the PCB vector slot must contain the address of a PCB other than the null PCB.

If the IPID represents a kernel thread, the high-order word of the IPID must fall within the numbers in the sequence vector entry corresponding to the process index: it must be greater than or equal to the contents of the base sequence number field and less than those of the next sequence number field. The ID of the kernel thread associated with that IPID is the difference between its sequence number and the base sequence number. Each multithreaded process has an associated KTB vector containing the addresses of its KTBs. For an IPID of a kernel thread to be valid, the entry for the kernel thread identified by that IPID must contain a KTB whose KTB$L_IPID matches the IPID of interest. Chapter 5 contains further information.

### 7.3.2 Extended PID

The EPID serves as a VMScluster-wide process identification. It is constructed from the IPID, which is unique across a local system but not across the nodes of a VMScluster. Figure 7.6 shows its layout.

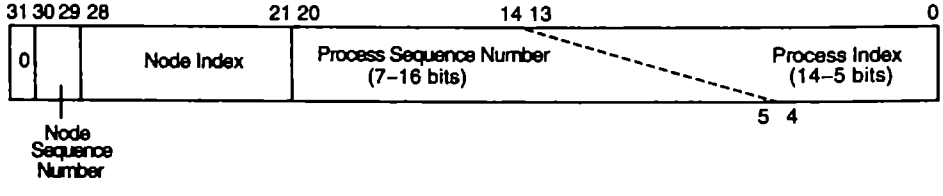



**Figure 7.6**
Layout of an Extended Process ID (EPID)

Its low-order 21 bits contain the IPID in two fields. The widths of these two fields vary, depending on the value of the SYSGEN parameter MAXPROCESSCNT. The first field, beginning at bit 0, contains the process index. The size of the field is computed at system initialization and stored in global location SCH$GL_PIXWIDTH. The second field contains the sequence number. Its size is 21 minus the size of the first field. In versions prior to OpenVMS Alpha Version 7.0, the maximum process index field was 13 bits; in OpenVMS Alpha Version 7.0, the field width has been increased by one bit, to 14 bits, allowing for a larger number of processes.

Bit 31 of the EPID is zero, preserving the rule that an EPID or IPID is never negative. The other ten high-order bits identify the VMScluster node. The node identification is similar to process identification in that it consists of an index into a node table and a sequence number that counts how many times the index has been reused. On a system that is not a VMScluster node, these bits are all zero.

After a system becomes a VMScluster node, the EPIDs of any existing processes must be updated with the node information, which comes from the node's cluster system identification (CSID). The low-order ten bits from the global location SCH$GW_LOCALNODE are inserted into the field PCB$L_EPID of each process and, if appropriate, into the field PCB$L_EOWNER.

## 7.4 SWAPPER'S ROLE IN PROCESS CREATION

After EXE$CREPRC has built a new process's PCB, KTB for its initial kernel thread, and if necessary, JIB, the swapper allocates physical memory for the new process. It calls SWP$SHELL_INIT, in module SHELLxK, to create and initialize the new process's page tables, PHD, kernel stack, and initial address space.

In OpenVMS Alpha Version 7.0, a process's page tables are not mapped in system space; they are only mapped in the page table space of the process's own virtual address space. Initializing another process's page tables is therefore not entirely straightforward.

SWP$SHELL_INIT temporarily maps into system space the physical page allocated for the new level 1 page table (L1PT) and initializes the level 1 page table entries (L1PTEs) that map system space level 2 page tables (L2PTs). It then performs a partial context switch so as to be executing with that L1PT but continuing to run on the swapper's kernel stack, which is in system space. From that context, able to access system space and the page table space of the new process, SWP$SHELL_INIT initializes the process-private level 2 page table (L2PT) that maps P0 and P1 space. It also initializes the P1 page table (P1PT) that maps the permanent part of P1 space by copying its initial contents from the shell.

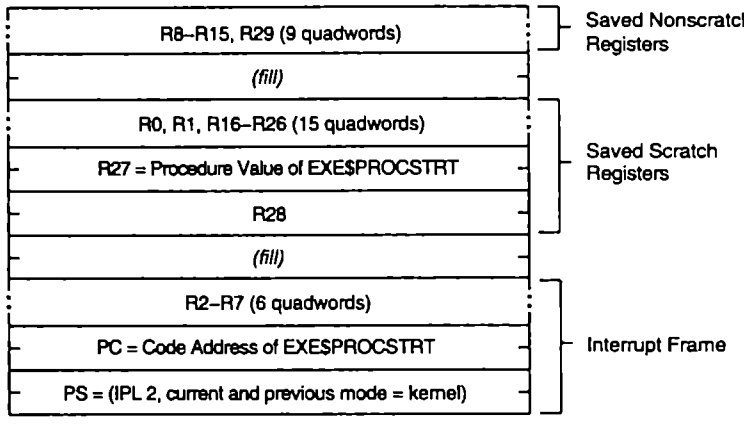



**Figure 7.7**
Initial Kernel Stack

The PHD is a complex data structure with a fixed part and a variable part. SWP$SHELL_INIT copies the fixed part of the PHD from the shell and uses various SYSGEN parameters to size the variable part, which consists of the BAK array, the working set list, and the process section table (PST). The BAK array is a quadword array with an entry for each page of PHD to record its backing store location during outswap. Because the PHD is mapped in both system space and the process's P1 space, SWP$SHELL_INIT must initialize both system page table entries (SPTEs) and P1 page table entries (P1PTEs) to map it.

SWP$SHELL_INIT creates the kernel stack for a new process's initial kernel thread partly by copying a prefabricated one from the shell. The initial contents of the prefabricated kernel stack, shown in Figure 7.7, are equivalent to the interrupt stack frame and saved integer registers found on the kernel stack of any noncurrent process (see Figure 2.10). The initial program counter (PC), R27, and processor status (PS) on this kernel stack are defined to cause the execution of the final routine involved in process creation (see Section 7.5). All other saved registers contain zeros.

The fixed part of the PHD includes a hardware privileged context block (HWPCB), initialized to define the privileged context for the initial kernel thread. The combination of the HWPCB and the information on the prefabricated kernel stack specifies the hardware context for the new process's initial kernel thread.

SWP$SHELL_INIT initializes several other required P1 pages from shell contents and locks some of them permanently into the process's working set.

The sections that follow describe these steps in more detail. Chapter *Memory Management Overview* contains information on page tables and the address translation algorithm, and Appendix *Size of System and P1 Virtual Address Spaces* discusses the layout of P1 space and the size of the PHD.

### 7.4.1 The Shell and Initial Process Pages

Conceptually, the shell represents the minimal process context necessary to execute kernel mode code that can flesh out the process more fully. The shell contains templates for most of a process's initial pages:

- Fixed part of its PHD
- P1PT that maps the permanent part of P1 space
- P1 pointer area
- Record Management Services (RMS) data area
- First page of kernel stack for the initial kernel thread

In addition to these pages, the shell contains SWP$SHELL_INIT.

SWP$SHELL_INIT initializes the previously listed pages by copying template contents from the shell and clearing the rest of the page.

It initializes other pages entirely at run time, as described in Section 7.4.3:

- The L1PT
- The L2PT that maps P0 and P1 space
- Any additional kernel stack pages

### 7.4.2 Preparation for Inswap of a New Process

The swapper takes the following steps in preparation for the inswap of any process:

1. It allocates physical memory for the process. In the case of any newly created process, SWP$GL_SHELLSIZ contains the number of pages to be allocated. Computed during system initialization as a function of SYSGEN parameters and the shell constants, this value is large enough to accommodate the pages listed in Section 7.4.1.
2. The swapper records the page frame numbers (PFNs) of the allocated pages in a quadword array called the swapper map. SWP$GL_MAP contains the starting address of the array.
3. It allocates a balance set slot, an entry in the system space array that contains the PHDs of resident processes.

   After allocating a balance set slot, the swapper examines PCB$L_WSSWP, the location of the outswapped process. A zero value in this field identifies a newly created process that must be initialized from the shell. To complete the configuration of the process and create the page table hierarchy, the swapper calls SWP$SHELL_INIT.

### 7.4.3 Configuring the New Process

SWP$SHELL_INIT's arguments include the addresses of the new process's PCB, the swapper map, and the HWPCB that the swapper uses temporarily to adopt the page tables of a process being swapped.

Running in kernel mode with the SCHED and MMG spinlocks held, initially in the swapper's process context, SWP$SHELL_INIT performs the following actions:

1. It calculates the address of the SPTE that maps the start of the new process's PHD. It copies the first entry from the swapper map into that SPTE, initializing it with the PFN allocated by the swapper, and sets its address space match (PTE$V_ASM) bit. This SPTE will map the fixed part of the PHD, BAK array, initial working set list, and initial PST.
2. It copies the fixed portion of the PHD from the shell to the first PHD page and clears the BAK array.
3. It copies the template working set list and clears the rest of the area reserved for the initial PST and working set list.
4. SWP$SHELL_INIT initializes an L1PT for the new process by taking the following steps:

   a. It gets the PFN of the physical page allocated for the new process's L1PT from the next entry of the swapper map and stores it in both the temporary HWPCB and the HWPCB of the new process's initial kernel thread.
   b. It maps the new process's L1PT in system space temporarily so that it can access it using system addresses.
   c. In that L1PT, it clears all L1PTEs lower than the one that will map the L1PT.
   d. It updates the PFN database for that PFN to reflect its use as the new process's L1PT. This includes zeroing PFN$W_PT_VAL_CNT to indicate that the page table maps one valid page, namely itself. This PFN database counter is based at −1. It initializes PHD$L_PTCNTVAL, the number of page table pages with valid pages, to 1 to indicate that one page table page maps a valid working set list entry (WSLE).
   e. It initializes the self-map L1PTE, the one that maps the L1PT, with the PFN; bits valid, fault-on-execute, and no-execute set; and protection allowing kernel mode reads and writes.
   f. It copies from the swapper process's L1PT the L1PTEs that map system space.

5. It calls MMG_STD$SWAP_PTBR, in module EXCEPTION, with the address of the temporary HWPCB passed by the swapper. MMG_STD$SWAP_PTBR stores the current kernel stack pointer in that HWPCB, switches to the context of that HWPCB, and returns.

   SWP$SHELL_INIT is now running on the swapper process's kernel stack in the context of that HWPCB, using the new process's L1PT. Because the new process's body pages are not yet connected to the page

tables that will map them, SWP$SHELL_INIT can access only system space and the new process's page table space. It is still holding the MMG and SCHED spinlocks.

6. SWP$SHELL_INIT initializes the L2PT that maps P0 and P1 space:

   a. It calculates the page table space address of the corresponding L1PTE.
   b. It initializes it with the PFN of the next swapper map entry; bits valid, fault-on-execute, and no-execute set; and protection allowing kernel mode reads and writes.
   c. It clears all the PTEs in the L2PT.
   d. It updates the PFN database for that PFN to reflect its use as an L2PT of the new process.
   e. It stores the PFN into the new process's PHD$L_L2PT_WSLX.
   f. In the L1PT's PFN database entry it increments PFN$W_PT_VAL_CNT to indicate that that page table maps another valid WSLE (its second).

7. SWP$SHELL_INIT initializes the level 3 page table (L3PT), or P1PT, that maps the permanent part of P1 space:

   a. It calculates the page table space address of the corresponding L2PTE.
   b. It initializes it with the PFN of the next swapper map entry; bits valid, fault-on-execute, and no-execute set; and protection allowing kernel and executive mode reads and kernel mode reads and writes.
   c. It clears the L3PT except for those level 3 page table entries (L3PTEs) that will map the kernel stack and the permanent part of P1 space.
   d. It initializes the L3PTEs that map the kernel stack for the new process's initial kernel thread and the stack's guard page. The number of L3PTEs initialized is based on the SYSGEN parameter KSTACKPAGES. The L3PTEs are initialized with information such as owner and protection but do not yet contain PFNs and do not have a set valid bit.
   e. It initializes the rest of the P1 space L3PT by copying information from the shell.
   f. It updates the PFN database for that PFN to reflect its use as an L3PT of the new process.
   g. It stores the PFN into the new process's PHD$L_L3PT_WSLX.
   h. In the L2PT's PFN database entry it increments PFN$W_PT_VAL_CNT to indicate that that page table maps a valid WSLE (its first), the L3PT.
   i. It increments the new process's PHD$L_PTCNTVAL to account for the L2PT as another page table that maps a valid WSLE.

8. SWP$SHELL_INIT gets the PFN from the next entry of the swapper map and maps that page into system space temporarily so that it can access it using system addresses. This page will be the highest addressed page of the initial kernel thread's kernel stack. SWP$SHELL_INIT copies the initialized part of the kernel stack from the shell template.
9. SWP$SHELL_INIT computes the sum of kernel stack pages and PHD P1 window pages and verifies that there are enough free PTEs in the shell's P1PT to map those pages, generating the fatal bugcheck INCON_SHELL if not.
10. Skipping past any other PFNs in the swapper map for kernel stack pages, it gets the PFN from the next entry of the swapper map and maps that page into system space temporarily. This page will be the P1 pointer page. SWPSSHELL_INIT copies the initialized part of this page from the shell template, clearing the remainder of the page.
11. SWP$SHELL_INIT gets the PFN from the next entry of the swapper map and maps that page into system space temporarily. This page will be the RMS process I/O data area.
12. It copies the initialized part of this page from the shell template, clearing the remainder of the page.
13. SWP$SHELL_INIT confirms that it has reached the end of the swapper map and, if not, generates the fatal bugcheck INCON_SHELL.
14. It stores the balance set slot index in the PHD. This value indexes the PHD reference count array and the process index array as well as the balance set slots. Chapter *Memory Management Data Structures* describes these arrays.
15. It stores the SYSGEN parameters that determine the default page fault cluster size and the default page table page fault cluster size in the PHD.
16. It requests the initial page file assignment for the new process and reserves enough pages in the page file for its PHD pages and the shell pages. It stores the page file and reservation count in the PHD.
17. It calculates and stores the offset to the end of the PST (PHD$L_PST_BASE_OFFSET).
18. It calculates and stores the longword offset from the beginning of the PHD to the beginning of the floating-point register and execution data (FRED) page area in PHD$L_FREDOFF.
19. SWP$SHELL_INIT initializes PHD$L_PTCNTMAX, the number of page table pages with nonzero entries, to the number of permanent P1PT pages copied from the shell.
20. The PHD page copied from the shell contains initial values for the three working set list longword index values (PHD$L_WSLOCK, PHD$L_WSDYN, and PHD$L_WSNEXT). SWP$SHELL_INIT adjusts the indexes to account for any additions to the permanent part of the working set such as extra kernel stack pages.

After altering the index to the dynamic portion of the working set (PHD$L_WSDYN), SWP$SHELL_INIT moves any dynamic WSLEs from their old location to the new location.

21. If any extra kernel stack pages were allocated, SWP$SHELL_INIT creates WSLEs for them.
22. SWP$SHELL_INIT updates the process working set list with the pages composing the beginning of the PHD (fixed portion, BAK array, working set list, and PST). In addition, it updates the PFN database records for the physical pages to indicate that these pages are active modified page table pages. It also records their backing store information, working set list offsets, page file, and PTE back pointers.
23. It records the assigned process page file as backing store information for the L2PT and P1PT pages.
24. OpenVMS Alpha Version 7.0 supports a new feature called a virtual region. A virtual region is a reserved area of process-private virtual address space. Virtual regions promote modularity within an application by allowing different components of the application to create address space in different virtual regions. OpenVMS Alpha Version 7.0 provides system services to define new regions in addition to the three process-permanent regions: P0 space, P1 space, and the default P2 region. Each region is described by a region descriptor entry (RDE). The three process-permanent RDEs are in the PHD.

    SWP$SHELL_INT records the address lower than the beginning of the PHD window as the first free P1 space address in PHD$PQ_P1_FIRST_FREE_VA. The rest of the P1 process-permanent RDE and all of the P0 RDE were initialized in the shell.
25. It calculates the number of available process-private PTEs in the sole existing P1PT and stores it in PHD$Q_FREE_PTE_COUNT.
26. It initializes the P2 process-permanent RDE to encompass all of P2 space.
27. SWP$SHELL_INIT gets the PFN from the swapper map corresponding to the P1 pointer page and maps the page into system space temporarily to perform additional initialization:

    a. It calculates the address of the P1 window to the PHD and stores it in the location corresponding to CTL$GL_PHD.
    b. It calculates the address of the working set list within the P1 window and stores it in the location corresponding to CTL$GL_WSL.
    c. It stores the PCB address in the location corresponding to CTL$GL_PCB.

28. It adjusts the address of the first free virtual address in P1 space (stored in the PHD at offset PHD$L_FREP1VA) to reflect the size of the PHD that is mapped into P1 space.
29. It initializes the PHD queues of RDEs to empty queues.

30. It initializes the WSEXTENT and WSAUTHEXTENT indexes to reflect the value of the SYSGEN parameter WSMAX, and the WSQUOTA and WSAUTH indexes to reflect the value of WSMAX or 65,536 pages, whichever is smaller. It initializes PHD$L_WSFLUID to the value of the SYSGEN parameter MINWSCNT. The end of the working set list (WSLAST) and the default count (DFWSCNT) initially reflect the value of the SYSGEN parameter PQL_DWSDEFAULT. PHD$L_WSSIZE is initialized to the value of PQL_DWSDEFAULT.
31. SWP$SHELL_INIT marks the PHD resident by setting bit PCB$V_PHDRES in PCB$L_STS.
32. SWP$SHELL_INIT returns control to the swapper's main inswap routine.

The swapper completes the remaining steps of the inswap operation. In particular, these steps include further initialization of the process's page tables and its HWPCB. As the final step, the swapper calls the scheduler routine SCH$CHSEP, in module RSE, to change the state of the new process's initial kernel thread to computable (COM) and possibly trigger a rescheduling interrupt. These steps are described in Chapter *The Swapper*.

## 7.5 PROCESS CREATION IN THE CONTEXT OF THE NEW PROCESS

The final steps of process creation take place in the context of the newly created process's initial kernel thread. Its initial register context is contained within the kernel stack copied from the shell. When the kernel thread becomes current, it begins execution at the saved PC in the kernel stack, the address of the routine EXE$PROCSTRT, in module PROCSTRT. The saved PS indicates kernel mode at IPL 2. Thus, the first code that executes in the context of a newly created process is the same for every process in the system.

### 7.5.1 Operation of EXE$PROCSTRT

When EXE$PROCSTRT begins execution, the PCB, initial kernel thread's KTB, and PHD have been created. In addition, information passed from the creator process has been copied into the PQB by EXE$CREPRC. The essential P1 space areas have been created and initialized (for example, kernel stack and P1 pointer area). EXE$PROCSTRT must create the rest of the permanent P1 space areas and copy PQB information into P1 space and the PHD (see Figure 7.8). EXE$PROCSTRT then prepares for and activates the image specified by the creator process.

EXE$PROCSTRT begins execution in kernel mode at IPL 2. Later segments of the code execute in executive mode and user mode. It performs the following steps:

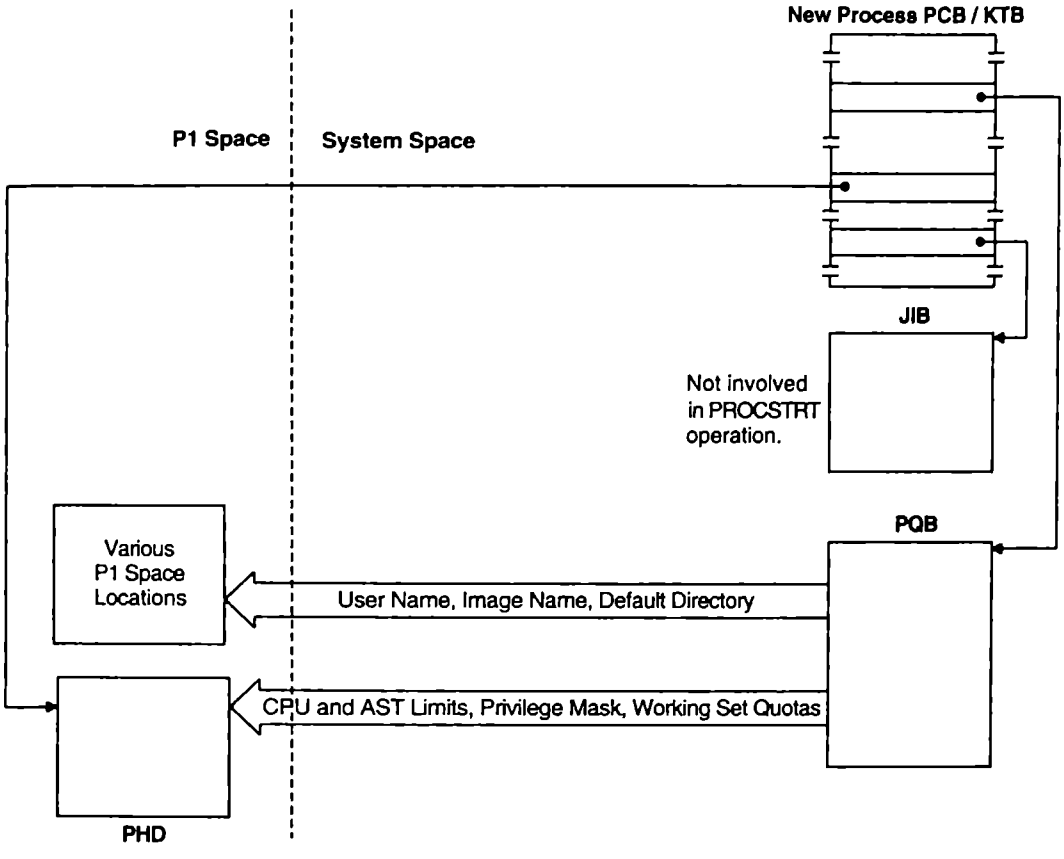



**Figure 7.8**
Removal of Process Parameters from the Process Quota Block

1. It obtains the PCB address from CTL$GL_PCB and the PQB address from PCB$L_PQB.
2. It initializes CTL$GL_CTLBASVA with the value in MMG$GL_CTLBASVA, which represents the initial low-address end of P1 space. The value in CTL$GL_CTLBASVA is updated with each expansion of process-permanent P1 space; it represents the boundary between process-permanent and image-specific P1 space.
3. It initializes CTL$GQ_PRCTERM_LISTHEAD to an empty list. This list supports the Process Termination Notification ($PRCTERM) system service. A process that has established a debugger connection with another process requests this service to receive notification through an AST that the other process has been deleted. Use of this service is reserved to Digital; any other use is unsupported.
4. It initializes several P1 cells related to multithreading support:
   - CTL$GL_KT_COUNT, the number of kernel threads, to the value 1

- CTL$GL_INITIAL_KTB, with the address of the PCB, which is overlaid by the initial kernel thread's KTB
- CTL$GL_INITIAL_PID, with the IPID of the current process
- CTL$GL_INITIAL_EPID, with the EPID of the current process

5. EXE$PROCSTRT initializes the P1 space locations CTL$GL_USRUNDWN and CTL$GL_USRUNDWN_EXEC with the addresses of the tables within the activated privileged library dispatch vector for kernel mode and executive mode image rundown routines. It initializes CTL$GL_GETMSG with the address of the start of the message vector, also in the activated privileged library dispatch vector. Chapter 8 contains details about the dispatch vector and rundown routines, and Chapter *Condition Handling*, about the dispatch vector and the message vector.
6. If the creator process requested image dump support, EXE$PROCSTRT propagates that flag to the PHD.
7. EXE$PROCSTRT initializes the kernel request packet (KRP) lookaside list (see Chapter *Pool Management*), forming the reserved address space into KRPs and inserting them on the list.
8. It moves the CPU time limit and the AST limit from the PQB to the PHD (see Table 7.3).
9. EXE$PROCSTRT initializes the working set list pointers in the PHD to reflect the quotas passed from the creator. It takes the minimum of the SYSGEN parameter WSMAX, the maximum working set size, and the number of potentially available physical pages to get maximum physical pages. It then enforces the following restrictions on quotas:
   - Working set quota must be less than or equal to 64K pages, the maximum size of a swap slot.
   - Working set extent must be less than or equal to the maximum physical pages.
   - Working set quota must be less than or equal to working set extent.
   - Working set default must be less than or equal to working set quota.
10. EXE$PROCSTRT copies the process's base priority to PHD$L_AUTHPRI and PCB$L_AUTHPRI. Saving the base priority enables a process without ALTPRI privilege to lower its base priority and later raise it as high as the original base priority.
11. It copies the process privilege mask from the PQB to the first quadword of the PHD (PHD$Q_PRIVMSK), the permanent privilege mask (CTL$GQ_PROCPRIV in the P1 pointer area), and the authorized privilege mask (PHD$Q_AUTHPRIV). Chapter 8 describes the use of each of these privilege masks.
12. It copies the default message flags to P1 space (CTL$GB_MSGMASK).
13. It saves the login time in CTL$GQ_LOGIN and CTL$GQ_PSTART.

14. EXE$PROCSTRT copies the minimum and maximum authorized security clearance records from the PQB to the PHD.
15. It initializes the following listheads as empty:

    - The deaccess audit pending queue
    - The objects rights block (ORB) lock database
    - The security object context queue
    - The Declared Resource Manager queue, used by the Digital Distributed Transaction Manager (DECdtm)
    - The three image activator listheads: image control blocks (IMCBs) representing activated images; IMCBs representing work in progress; and the IMCB lookaside list (see Chapter 8)
    - The queue in the PCB of CWPSSRVs representing CWPS requests in progress

      Processes are visible and can be manipulated clusterwide. CWPS supports system services in implementing this feature (see Chapter 6).
    - The process scan queue in the PHD

      The $PROCESS_SCAN system service uses this queue to maintain its search context (see Chapter 6).
    - The list of queue manager messages

16. EXE$PROCSTRT creates P1 virtual address space for the channel control block table and the process allocation region. It reserves P1 space in which the image activator can map data sections of images installed with shareable linkage (see Chapter 8). Data sections from such an image must be mapped at the same virtual address in each process.

    Appendix *Size of System and P1 Virtual Address Spaces* describes these areas and the SYSGEN parameters that affect their size. EXE$PROCSTRT records the address of each area and updates the process-permanent boundary address in CTL$GL_CTLBASVA with the new, lower address.
17. It calls RM$INIT, in module RMSRESET, to create and initialize the process and image I/O segments (see Appendix *Size of System and P1 Virtual Address Spaces*).
18. It allocates and initializes space from the P1 allocation region for the process logical name hash table. It computes the size of the table based on the number of hash table entries, but in no case does it allow the size of the hash table to exceed half the size of the P1 allocation region. EXE$PROCSTRT also allocates space for the process-private logical names and tables that it will create. Chapter *Logical Names* describes the logical name data structures and their use.
19. EXE$PROCSTRT then allocates space from the P1 allocation region for the process-private logical name table cache. It formats the space into a lookaside list of logical name cache entries.

20. It initializes the process directory logical name table, LNM$PROCESS_DIRECTORY, and the process logical name table and inserts them into the hash table.
21. EXE$PROCSTRT creates the logical name table logical names LNM$PROCESS, LNM$GROUP, and LNM$JOB. It inserts them into the hash table and into LNM$PROCESS_DIRECTORY.
22. Using the equivalence strings and logical name attributes from the PQB, EXE$PROCSTRT creates the logical names SYS$INPUT, SYS$OUTPUT, SYS$ERROR, TT, and SYS$DISK.
23. EXE$PROCSTRT copies the image name from the PQB to the image header buffer for subsequent use by the image activator (see Chapter 8).
24. EXE$PROCSTRT copies the default directory string, if one exists, from the PQB to P1 space. It also copies the two sets of names for the CLI and the command table, one for this process and another for its subprocesses, from the PQB to P1 space.
25. It copies the $CREPRC flags, user authorization file (UAF) flags, and debug flags from the PQB to P1 space flags.
26. It copies the user name from the JIB and the account name from the PQB into the P1 pointer area.
27. If the process is not a subprocess, EXE$PROCSTRT creates the job and group logical name tables. (If the process is a subprocess, then the tables already exist.) Because multiple processes access the tables, they must be in system space. EXE$PROCSTRT allocates space for the tables from paged pool. It locks the logical name table mutex for write access and holds it while accessing the shareable logical name hash table. Chapter *Synchronization Techniques* describes mutexes.

    EXE$PROCSTRT initializes the two tables and inserts the job table into the shareable logical name hash table. It attempts to do the same with the group table. However, the group table may have already been created by some other process with the same UIC group number. If this is the case, the new table is unnecessary and EXE$PROCSTRT deallocates it back to paged pool. Otherwise, EXE$PROCSTRT inserts the group table into the shareable logical name hash table. In either case, it unlocks the logical name table mutex.
28. EXE$PROCSTRT deallocates the PQB by inserting it on the PQB lookaside list (see Chapter *Pool Management*).
29. It calls EXE$JOIN_SYSTEM_DOMAIN, in module RESOURCE_DOMAIN_OBJECT, to join the system resource domain so that executive and file system code running in the process's context can use system locks. (Chapter 4 provides further information on system locks.)
30. It calls MMG$IMGRESET, which resets PHD$L_WSLAST, the pointer to the end of the working set list. MMG$IMGRESET also lowers IPL to 0, making it possible for the process to be deleted.

Another, more philosophical, interpretation is that at this point in the creation of a process, something exists that is capable of being deleted, a full-fledged process; all critical data structures associated with the process have been created and are in a consistent state.

31. EXE$PROCSTRT initializes the shareable image list for the Address Relocation Fixup ($IMGFIX) system service to point to a dummy element. This system service is described in Chapter 8.
32. EXE$PROCSTRT dispatches to F11X$INIT_XQP, in module [F11X]INIFCP, initialization code within the XQP image. The initialization code requests the Expand Program/Control Region ($EXPREG) system service to create a process-private copy of the XQP impure area and space for the XQP's private kernel stack. The code then updates CTL$GL_CTLBASVA. After performing other Files-11 initialization, it returns to EXE$PROCSTRT.
33. At this point, EXE$PROCSTRT has moved all the information from the creator to the context of the new process and is now ready to activate the image that will execute in the context of the new process. It must change mode to executive to request the image activator, which is an executive mode system service.

    EXE$PROCSTRT changes access mode to executive by calling EXE$REI_INIT_STACK, in module REI.
34. Running in executive mode, EXE$PROCSTRT requests the Image Activator ($IMGACT) system service to set up the page tables and perform the other steps necessary to activate the image specified by the $CREPRC request that caused this process to be created. Image activation is described in Chapter 8.
35. EXE$PROCSTRT declares an executive mode exit handler, EXE$RMSEXH. This handler will be called when the Exit ($EXIT) system service is requested from executive mode, which usually happens when the process is deleted. When called, EXE$RMSEXH calls SYS$RMSRUNDWN for each open file.
36. EXE$PROCSTRT stores the procedure value of a dummy CLI call back routine in location CTL$AL_CLICALBK. If an image that was activated from EXE$PROCSTRT attempts to communicate with a nonexistent CLI, the dummy CLI call back routine will return the error status CLI$_INVREQTYP.
37. EXE$PROCSTRT changes access mode to user by calling EXE$REI_INIT_STACK.
38. It clears the frame pointer (R29), guaranteeing that the search of the user mode stack for a condition handler by the exception dispatcher will terminate (see Chapter *Condition Handling*).
39. EXE$PROCSTRT sets up an initial stack frame on the user mode stack by calling an inline procedure:

```
        CALLS   #0, 260$
        .
        .
260$:   .CALL_ENTRY INPUT=<R8,R9,R10>
        MOVAB   R9, (FP)
        .
        .                                   ;Procedure code
```

40. EXE$PROCSTRT establishes the catch-all condition handler, EXE$CATCH_ALL, as the condition handler for this stack frame and also as the last chance condition handler for user mode. The purpose and action of this handler are discussed in Chapter *Condition Handling*.
41. EXE$PROCSTRT requests the $IMGFIX system service to perform address relocation for the image.
42. An argument list that is nearly identical to the one used by a CLI (see Chapter 11) is built on the stack. This argument list allows an image to execute in the same manner regardless of whether it was activated from EXE$PROCSTRT or from a CLI.
43. EXE$PROCSTRT determines whether the process was created with the hibernate STSFLG argument. If the PCB$V_HIBER bit in PCB$L_STS is set, it requests the Hibernate ($HIBER) system service. EXE$PROCSTRT will continue when the process is awakened.
44. It calls the image at its initial transfer address (see Figure 8.26). The image executes. When done, the image can return to EXE$PROCSTRT or request the $EXIT system service directly or indirectly through compiler-supplied means. If the image returns to EXE$PROCSTRT and if the process was created with the hibernate STSFLG argument, EXE$PROCSTRT places the process's initial kernel thread back into hibernation. When it is awakened, EXE$PROCSTRT calls the image again at its initial transfer address. An effect of this implementation is that the image is not exited or run down and no exit handlers (user-declared or system-declared, such as EXE$RMSEXH) are called. Writable image data initialized at compile or link time is not reinitialized but rather retains values stored during the previous image execution. Shareable images' initialization routines are not called again, although the main image's initialization routine is.

    If the process was not created with the hibernate flag, EXE$PROCSTRT requests the $EXIT system service itself.

    In general, there is no difference between an image terminating by returning to EXE$PROCSTRT or by requesting the $EXIT system service. If, however, the process was initially created with the hibernate flag so that it could be awakened and reentered, the image must return to EXE$PROCSTRT rather than terminate by requesting the $EXIT system service.

## 7.6 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIB]JIBDEF.SDL
[LIB]PCBDEF.SDL
[LIB]PQBDEF.SDL
[SYS]PROCSTRT.MAR
[SYS]SHELL.MAR
[SYS]SYSCREPRC.MAR

# 8 Image Activation and Exit

> I would have you imagine, then, that there exists in the mind of man a block of wax . . . and that we remember and know what is imprinted as long as the image lasts; but when the image is effaced, or cannot be taken, then we forget or do not know.
>
> Plato, *Dialogs, Theaetetus* 191

Programs are compiled and linked to produce images, which execute in the context of a process. Before an image can execute, the OpenVMS Alpha executive must locate the image file, open and read it, map the image into address space, and resolve address references in the image. Image activation is the combination of these steps.

At image exit the executive must call exit handlers declared by itself or by the image. In a process with a command language interpreter (CLI), multiple images can execute one after another. When one image exits, the executive must eliminate all its traces and deactivate multithreading so that the next image can begin execution with no side effects from the execution of the previous image. This is referred to as image rundown.

This chapter details these activities and also describes the initialization and use of the various privilege masks maintained for each process. It describes the following system services related to image activation and exit:

- Image Activate ($IMGACT)
- Address Relocation Fixup ($IMGFIX)
- Image Startup ($IMGSTA)
- Declare Exit Handler ($DCLEXH)
- Exit ($EXIT)
- Rundown ($RUNDWN)

Use of the $IMGACT, $IMGFIX, $IMGSTA, and $RUNDWN system services is reserved to Digital. Any other use is unsupported.

## 8.1 IMAGES

An image is a file that contains binary code and data, and that can execute on an OpenVMS Alpha system. The linker builds an image from one or more object modules produced by a language processor such as a compiler.

This section describes the structure of an image and the various image types. It summarizes the mapping of image sections into virtual address space.

### 8.1.1 Program and Image Sections

Language processors create program sections and define their attributes. A program section (PSECT) is a named piece of a program with a given set of attributes. PSECT attributes control various characteristics of the area of memory described by the PSECT. For example, they specify whether the PSECT is writable (WRT or NOWRT) and whether it contains code or data (EXE or NOEXE).

Programmers working in high-level languages typically have little direct control over the PSECTs created by the language processor. Medium- and low-level languages provide programmers with more control over PSECT creation. However, most PSECT attributes can be altered at link time through the use of a linker options file.

When compiling a source module, a language processor generates records in the object module that tell the linker which PSECTs are present in the module and what attributes they have. In building an image from a set of object modules, the linker combines PSECTs with similar attributes to form image sections. An image section is a group of PSECTs with the same significant attributes, thus requiring a contiguous set of virtual pages with identical properties. In this sense, it is the unit of virtual memory allocation for an image.

The PSECT writable attribute determines the protection on the virtual pages into which the resulting image section will be mapped when the program is run. For example, an access violation would result at run time if a program's code tried to write to a NOWRT portion of the program.

An image can refer to locations within itself or within another image. When compiling a source module, an OpenVMS Alpha language processor creates a special PSECT called a linkage section, which contains descriptors of the module's procedures and address constants that a procedure uses to access static storage, external procedures, and variables. This address data must be adjusted when the image is loaded into memory to reflect the actual locations assigned by the image activator.

When building an image, the linker builds a special image section called a fixup section, which describes the locations in the image that contain addresses that must be adjusted after all the image sections have been assigned starting addresses. The adjustment to an intra-image address reference is called a relocation. The adjustment to a reference from one image to another, an inter-image reference, is called a fixup.

For each image section, the linker builds into the image a data structure called an image section descriptor that contains information such as the amount of memory in the section, section attributes, and so on.

This information is used by the image activator to map the image. In general, virtual address space is broken into sections. Although the unit of address translation is the page, the section is the unit by which virtual address

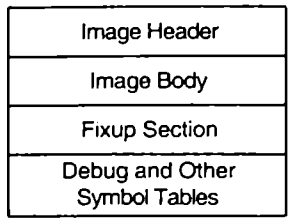



**Figure 8.1**
Organization of an Image

space is managed. A section is a group of contiguous virtual pages with the same characteristics.

### 8.1.2 Image File Structure

An image consists of several variable-sized pieces, the first of which is the image header. The image header contains information about the image such as its name and the manner in which it was linked. It also contains the image section descriptors, which together describe the virtual address space requirements of the image.

The image header is followed by the image body, the actual program code and data; by a fixup section; and, optionally, by symbol table information. The image body is made up of image sections of varying characteristics. Strictly speaking, the fixup section is part of the image body. Figure 8.1 shows the organization of an image.

Chapter 9 describes the organization of a translated image, which includes additional, special-purpose image sections. Section 8.2.1.1 describes the image header, Section 8.2.1.3 the image section descriptors, and Section 8.4.2 the fixup section.

### 8.1.3 Image Characteristics

An image's characteristics at run time are determined by the manner in which it was linked, whether it was installed, and if so, the qualifiers specified to the Install utility. The image can be categorized by whether it is

- An executable or a shareable image
- Installed, or made known to the system (and to what degree)
- Shared
- Permanently memory-resident
- Protected

The way in which an image is linked determines whether it is executable or shareable. An executable image has a unique entry point called a transfer address and can be run in response to a Digital command language (DCL) RUN command. An executable image can contain both code and data from

object modules produced by the various language compilers, and connections to other, separately linked images called shareable images. An executable image can be linked with multiple shareable images, which themselves can be linked with other shareable images.

A shareable image is a collection of procedures linked to form a unit. The linker creates a symbol vector for the shareable image, essentially a table locating procedure entry points and data items available for external modules to link against. The procedures can be called by code in an executable image or in another shareable image, but the shareable image is not directly executable. One special kind of shareable image is an executive image, described in detail in Chapter *The Modular Executive*.

In this chapter, the term *main image* refers to the primary controlling executable image invoked by a user through the RUN command. Activating a main image results in activating any shareable images with which it was linked and any with which they were linked. Furthermore, after a main image has begun to execute, code within it or one of its shareable images can call the Run-Time Library procedure LIB$FIND_IMAGE_SYMBOL, which uses the $IMGACT system service to activate yet another shareable image.

The Install utility causes an image to become known to the system. Both executable and shareable images can be installed. Qualifiers on the INSTALL command determine the detail known about the image (the amount of information stored about it) and the amount of preprocessing done to expedite later image activation.

A protected image contains one or more image sections to be mapped into pages whose protection prohibits user and supervisor mode write. If all image sections are to be protected, the image is linked with the qualifier /PROTECT. Alternatively, a subset of image sections can be specified with the PROTECT link option. To be activated, a protected image must have been installed /PROTECTED. A completely protected image may not make calls to routines in shareable images other than system services or executive routines. One common type of protected image is a privileged shareable image, which typically contains user-written system services or inner mode rundown routines.

### 8.1.4 Uninstalled Images and Address Space

While actual memory allocation and address translation occur in units of pages, the unit by which virtual address space is managed is the section.

In building an executable image, the linker tentatively assigns a virtual address range to each image section. In building a shareable image, the linker establishes the relations among the sections that make up the image but does not assign a base address to the first section.

When building an executable image from object modules linked with one or more shareable images, the linker assigns virtual address ranges

to the executable image's sections but not to those of the shareable images. Having their address ranges assigned at activation time makes it possible for an image linked with one version of a shareable image to be activated with a different version whose size is different. This feature is particularly significant for providing upward compatibility and portability of images.

The page size the linker uses in creating an image affects the address boundary on which each image section begins. The linker qualifier /BPAGE specifies the page size, also known as the grain, whose default value represents a page size of 64 KB. Although all current Alpha CPUs support a page size of 8 KB, an image linked to a 64 KB grain could execute on any future Alpha CPU with a page size larger than 8 KB.

An image linked to an 8 KB grain has image sections beginning at 8 KB boundaries and thus may not be able to run on a CPU with a larger page size without relinking. The only disadvantage to linking with a 64 KB grain is the likelihood of empty page table entries (PTEs) between image sections. By default virtual page 0 is no-access and the first image section begins on the address boundary determined by the grain.

Conceptually, the simplest type of image activation is the activation of an executable image that has not been installed. Basically, the image activator creates a process-private P0 space section for each image section. It uses the address range assigned by the linker.

The next simplest type is the activation of an executable image linked with a shareable image, neither of which has been installed. The image activator creates a process-private P0 space section for each image section in the executable image, using the address space assigned by the linker. The image activator then creates a process-private P0 space section for each image section in the shareable image. The first image section of the shareable image is based at the first available virtual page following the executable image, disregarding the address boundary determined by the grain. The remaining image sections are based to preserve their relative positions as established by the linker.

Section 8.1.6 describes the address space assignments made for installed images. Section 8.1.7 provides an example of the activation of an executable image linked with multiple installed and uninstalled shareable images, including some with shareable address data.

### 8.1.5 Install Utility

Images are installed for several reasons:

- To accelerate image activation
- To enable sharing of the image by multiple processes
- To provide temporary privileges to users of the image in a controlled fashion

- To support user-written system services
- To be made permanently memory-resident

Some images must be installed in order to execute, because of the manner in which they are linked. Some images must be installed in order to be called from a main image installed with privileges. Images can also be linked specifically to take advantage of particular installation options, such as resident code.

The Install utility preprocesses an image and manages the known image database (also called the known file database) to describe images that are installed. Each installed image has an entry in this database, maintained in a known file entry (KFE), but the amount of information maintained is determined by the Install qualifiers:

- /OPEN. Every installed image is known; that is, it can be opened by its file ID rather than by its file name, saving the overhead of a file lookup. Installing an image /OPEN saves more overhead; it causes a window control block (WCB) and a file control block (FCB) to be created and kept for the file, thus maintaining the location of the file's disk blocks and its access control information in memory. As a result, Record Management Services (RMS) $OPEN requests during subsequent image activation are essentially null operations.
- /HEADER_RESIDENT. RMS scans the known image database whenever a file is opened with the known file option. If an image was installed /HEADER_RESIDENT, its image header (and thus, its virtual memory mapping information) is stored in paged pool. Keeping the header resident saves the additional read operations otherwise required to read it into memory every time the image is activated.
- /PRIVILEGED. An executable image that requires enhanced privileges but that must execute in nonprivileged process context (such as SET or SHOW) can be installed with the /PRIVILEGED qualifier. When such an image is activated, the process gains enhanced privileges temporarily. (All shareable images activated with the privileged image must also have been installed.) The enhanced privileges are removed when the image is run down.
- /SHARED. Both shareable and executable images can be installed /SHARED. Installing an image /SHARED causes the Install utility to identify the image's shareable sections, such as code and read-only data, and map them as permanent global sections. Processes activating the image map the global sections into their own address spaces and share a single copy in memory rather than having to obtain a private copy from the image file on disk.

  The Install utility's convention for naming the global sections it creates is to append the installed image's KFE address and the number of the image section to the string INS$. For example, INS$811998D0_002 identifies a global section created for the second image section descriptor of the image

whose KFE resides at $811998D0_{16}$. Thus, the global section name identifies both the image file and the image section.

In earlier versions of OpenVMS, global section names were a concatenation of the image file name and a numeric value that began at _001 and was incremented with each section created for the image. If the logical name INSTALL$FALLBACK is defined, the Install utility creates global section names using this former algorithm.

Consider, for example, installing an image named TEST that consists of six image sections, with image sections 2 and 4 to be mapped as global sections. Given a KFE address of $811998D0_{16}$, the current convention creates two global sections named INS$811998D0_002 and INS$811998D0_004. The earlier convention creates two global sections named TEST_001 and TEST_002.

- /PROTECTED. If an image is installed /PROTECTED, all shareable images with which it was linked must be installed. Other restrictions apply as well; the image activator, for instance, specifies that RMS use only executive or kernel mode logical names to translate the image name and its descendants' image names.

  An image must be installed /PROTECTED/SHARED to use a privileged library vector or a system service intercept vector. A user-written system service, for instance, would be installed /PROTECTED/SHARED.
- /RESIDENT. The Install utility treats the /RESIDENT qualifier as /RESIDENT/SHARED/OPEN/HEADER_RESIDENT.

  The Install utility can load shareable image sections, such as code or read-only data, from shareable or executable images into a granularity hint region. Sections loaded in this manner are made permanently resident, and references to addresses in these sections are likely to be quicker as a result of reduced translation buffer overhead (see Chapter *Memory Management Overview*).

  The image can be installed with code, data, or both resident; code sections are stored in the resident image code region, and read-only data sections in the resident image data region. The Install utility's default action is to make code sections resident and map shareable read-only data sections as global sections.

  An image installed /RESIDENT must have been linked with the appropriate /SECTION_BINDING qualifier, and sufficient space must be available in the desired granularity hint region. The image's KFE receives an additional structure containing adjusted image section mapping information for later use by the image activator.
- /SHARE=ADDRESS_DATA. Specifying the qualifier /SHARE=ADDRESS_DATA in addition to /RESIDENT when installing a shareable image causes the Install utility to preactivate the image. It assigns any image section not mapped in a granularity hint region to a permanent location in P1 space, so all processes activating the shareable image will map its sections in the

same system or P1 space addresses. Therefore, address relocation and most address fixups for the shareable image need only occur once, when the image is installed, rather than each time the image is activated.

Relocations and fixups primarily alter the linkage section, which contains address data and is normally mapped as a private section for every process activating the shareable image. Because the relocations are performed by the Install utility rather than by the image activator, the linkage section can instead be mapped as a permanent global section shared among all processes activating the image.

To successfully install with shareable address data, a shareable image can only link against shareable images that will also be installed with shareable address data. It must be linked with the appropriate /SECTION_BINDING qualifier.

Very frequently accessed shareable images are good candidates for this type of installation because it speeds up image activation and saves CPU time and page faults. Section 8.4.10 describes shareable address data in detail, and Figure 8.3 shows a shareable image installed resident with a shareable linkage section.

### 8.1.6 Installed Images and Address Space

When an image is installed /SHARED, the Install utility creates a system global section for any shareable image section, generally the code and read-only data sections. The image activator later maps a range of process address space to the global section. The Install utility and the image activator must use the same criteria to decide which sections in the image are shareable.

- Executable image installed /SHARED. For an executable image installed /SHARED, the image activator bases both private and shareable image sections using the P0 starting addresses assigned by the linker. The image activator creates a process-private P0 space section for each nonshareable image section but maps shareable sections to the global sections created by the Install utility. If the executable image is linked with a shareable image, the first image section of the shareable image is mapped beginning at the first available virtual page following the executable image, disregarding the address boundary determined by the grain.
- Shareable image installed /SHARED. A shareable image is activated only as a result of the activation of an executable image linked against it. For a shareable image installed /SHARED and activated with an executable image, the first image section is based at the first available virtual page following the executable image. The remaining image sections, private or shareable, are based to preserve their relative positions as established by the linker. The image activator creates a process-private P0 space section for each nonshareable image section but maps shareable sections to existing global sections.

When an image is installed /RESIDENT, the Install utility loads each resident section into the appropriate granularity hint region. It creates a data structure for each resident section and records the section's system space address there. It creates system global sections for other shareable image sections as usual.

To be installed in this manner, the image must have been linked with the appropriate /SECTION_BINDING qualifier. The /SECTION_BINDING=CODE qualifier prevents the linker from replacing jump instructions with branch instructions when the target crosses an image section boundary. This optimization would introduce dependencies in the code sections on their relative positions.

The /SECTION_BINDING=DATA qualifier causes the linker to check for dependencies in the nonshareable sections on their relative locations. If there are no dependencies, the image's nonresident sections can be mapped to reduce holes in the address space. The resulting elimination of address space holes is called compression. The term *compressed data sections* describes the newly adjacent sections. Note that the data is intact; it is the virtual holes that are compressed.

Activating an image installed /RESIDENT is an extension of activating an image installed /SHARED. When the image is activated, the image activator creates a process-private P0 space section for each nonshareable image section. For shareable sections, it either maps a range of process address space to an existing global section or finds the resident section in system space using the data structure created by Install.

- Executable image installed /RESIDENT. The nonresident image sections of the executable image are normally based at the P0 starting addresses established by the linker. However, if the image's data sections are compressed, the image activator overrides the base addresses established by the linker and bases each nonresident section at the next available virtual page, eliminating holes in the address space.
- Shareable image installed /RESIDENT. For a shareable image installed /RESIDENT and activated with an executable image, the first image section is based at the first available virtual page following the executable image. Multiple shareable images activated in succession establish base addresses at the next available virtual page.

  If the shareable image's data sections are compressed, the image activator bases each nonresident section at the next available page, overriding the relative positioning established by the linker and eliminating holes in the address space. Otherwise, it preserves the relative positions established by the linker.
- Shareable image installed /SHARE=ADDRESS_DATA/RESIDENT. A resident shareable image installed with shareable address data is activated when an executable image that links against it is activated. The base

address of each image section has been predetermined, and the image activator obtains each from a data structure created by the Install utility.

### 8.1.7 Image Mapping Example

The image activator maps the sections of an image into virtual address space based upon their characteristics. The System Dump Analyzer's executable image, SDA.EXE, is used here as an example to illustrate some of the concepts of image mapping. This is a simplification of a complicated activation, involving an executable image, an uninstalled shareable image, two shareable images installed /SHARED, and multiple shareable images installed /RESIDENT/SHARE=ADDRESS_DATA.

SDA.EXE is an executable image linked only with SDA$SHARE.EXE, a shareable image that is not normally installed. SDA$SHARE, however, is linked with several images installed resident with shareable linkage sections, as well as with SCRSHR.EXE and SMGSHR.EXE, shareable images installed /SHARED, and with SYS$BASE_IMAGE and SYS$PUBLIC_VECTORS. In order for SDA to execute, this entire tree of images must be activated. Figure 8.2 shows SDA.EXE and some of the shareable images with which it is linked mapped into a process's virtual address space.

Because the SDA.EXE executable image is linked with a default /BPAGE qualifier value, each of its image sections begins on a 64 KB ($10000_{16}$ byte) boundary. Each image section is activated at the base address determined by the linker. By default virtual page 0 is no-access. Thus, SDA's first image section is mapped beginning at $10000_{16}$. This image section's size is 512 bytes, or one pagelet. On current Alpha systems, which have an 8 KB ($2000_{16}$) page size, this section occupies an entire virtual page (equivalent to 16 pagelets). Note that this one-pagelet image section occupies a full virtual page, but the virtual page is only partially backed by the image file (see Chapter *Memory Management System Services*).

A number of P0 page table entries (P0PTEs) map null pages between the end of the first section and the beginning of the second, which also begins on a 64 KB boundary. There are null pages, or holes, between the end of most sections and the beginning of the next on current Alpha CPUs. The linker creates SDA.EXE's fixup section at the high-address end of the image, beginning on a 64 KB boundary.

SDA.EXE is linked with SDA$SHARE.EXE, which in turn is linked with many shareable images, most not shown in Figure 8.2. Despite the fact that SDA$SHARE.EXE was linked with a grain of 64 KB, the image activator maps it beginning at the next available page, $32000_{16}$, although that address is not a 64 KB boundary. This avoids one hole in the virtual address space, decreasing the number of null P0PTEs. This mapping is possible because the linker does not assign base addresses to shareable images. The individual image sections of SDA$SHARE.EXE must maintain their rela-

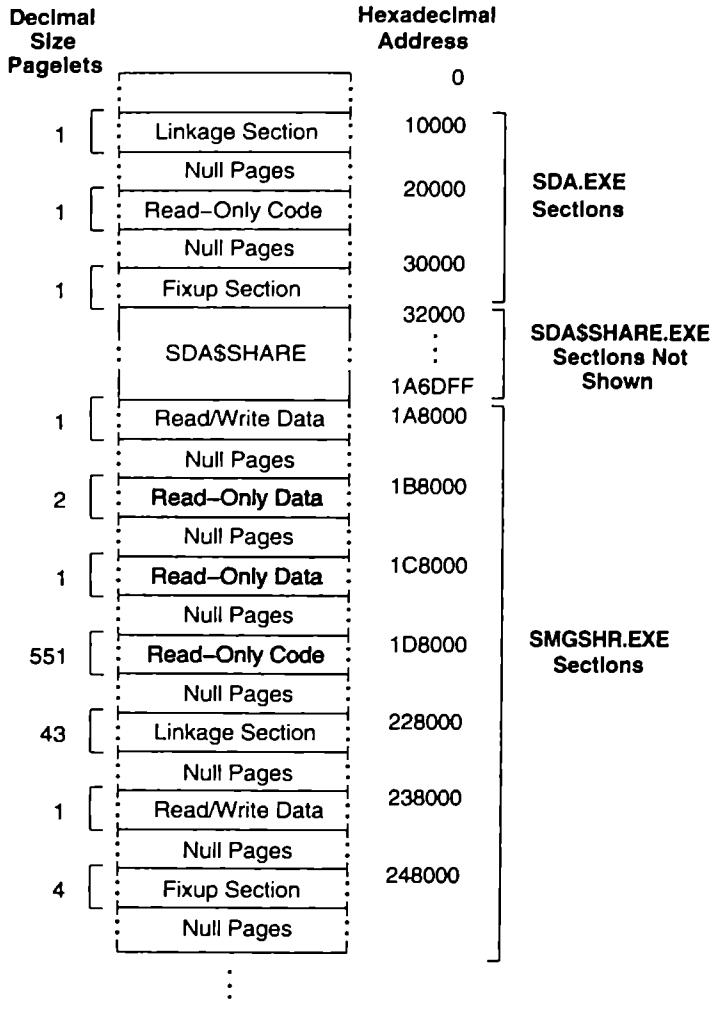



**Figure 8.2**
Mapping of an Executable Image with Shareable Images

tive position to one another, however, and holes between its sections still exist.

On the system from which the example was taken, the shareable image SDA$SHARE.EXE was not installed. The image activator therefore created a process-private P0 section to map each of its image sections.

However, because SMGSHR.EXE was installed /SHARED, the Install utility created a global section from each of its shareable sections. When SMGSHR.EXE is activated, its global sections are mapped into P0 space and process-private P0 sections are created for its copy-on-reference nonshareable sections. The shaded areas in Figure 8.2 represent global sections.

Note that linkage and fixup sections are not normally shared. Each contains addresses that are a function of the address space assigned to the image in this activation in this process; linked and activated with a different main

image, the same shareable image is likely to be assigned to a different address range. Unless the image is installed /RESIDENT/SHARE=ADDRESS_DATA, causing every process to reference it in the same address range, an image section in a shareable image with relocated or fixed up data is not shareable. For example, a symbol vector, which consists entirely of relocated data, cannot be shared. Thus, an image section that contains a symbol vector is copy-on-reference.

As shareable images, SDA$SHARE.EXE and SMGSHR.EXE contain relocation information in their fixup sections that enables the image activator to adjust their intra-image references to reflect their assigned address space. SDA.EXE's fixup section contains information that enables its references to SDA$SHARE.EXE to be fixed up to reflect the address space assigned to SDA$SHARE.EXE's sections. SDA$SHARE.EXE's fixup section contains information that enables its references to SMGSHR.EXE to be fixed up to reflect the address space assigned to SMGSHR.EXE's sections.

SDA$SHARE.EXE is also linked with the shareable image LIBOTS.EXE, which in this example has been installed /RESIDENT/SHARE=ADDRESS_DATA. The Install utility mapped LIBOTS.EXE's code section into the system space resident image code region, created global sections for its linkage section and its shareable read-only data section, and reserved P1 address space for these global sections and for each copy-on-reference section. Although LIBOTS.EXE is linked to allow its shared data sections to be installed resident, the system in this example has no space in the resident image data section; instead, the Install utility created a global section for the read-only data.

LIBOTS.EXE's linkage section is created by the compiler as a writable copy-on-reference section, that is, nonshareable, because it is the target of relocation and fixup operations. But because each process activating LIBOTS.EXE will find its resident code in the same system address space, and map its global and copy-on-reference sections into the same P1 address spaces, fixups applied to the linkage section resolve to the same addresses for each process. Thus, the Install utility can process most relocations and fixups at installation time, and the linkage section can become shareable.

Note that the linker options file used for LIBOTS.EXE collects the symbol vector and the linkage section into the same image section. Thus, the majority of fixups target this image section, and few fixups target process-private sections. Fixup writes to process-private sections and protection change fixups cannot be preprocessed by the Install utility. For such fixups, it creates a subset, compressed fixup section in system space. These remaining fixups are processed after the image is activated.

If LIBOTS.EXE were installed /RESIDENT rather than /RESIDENT/SHARE=ADDRESS_DATA, no global section would be created for its linkage section. The global section created for its shareable read-only data section would be mapped into P0 space rather than P1 space, and process-private P0 sections would be created to map its copy-on-reference nonshare-

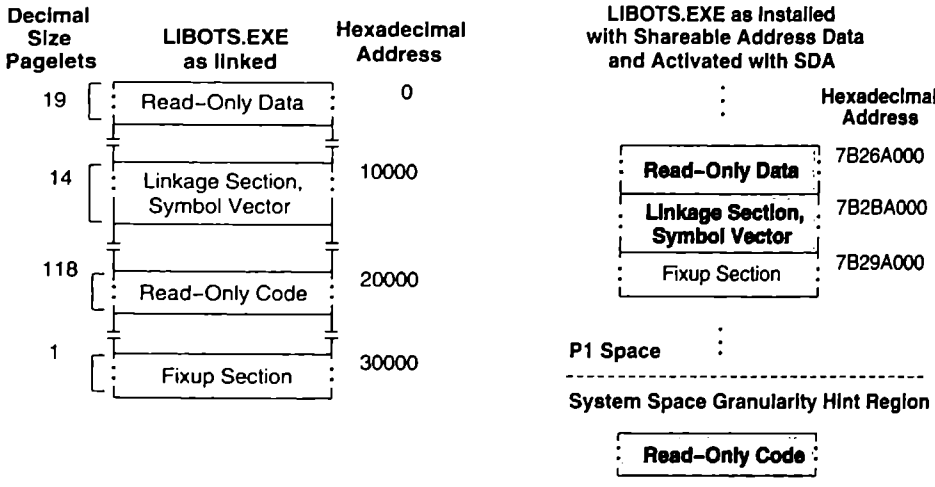



**Figure 8.3**
Mapping of an Image Installed Resident with Shareable Address Data

able sections, which would include the linkage section. No fixups would be performed until image activation time, when the linkage section would be faulted in and modified to reflect the assigned P0 address space.

The combined actions of the Install utility and the image activator result in image sections that are not necessarily in the same locations relative to each other as they are in the linked image. Figure 8.3 contrasts the relative positions of LIBOTS.EXE's sections in the linked image and in the activated image. The shaded areas in the figure represent shared sections. This figure demonstrates data compression, the elimination of address space holes. Note that the data is intact; it is the virtual holes that have been compressed. When LIBOTS.EXE's intra-image references are relocated and its inter-image references fixed up, and when references to it from other images are fixed up, the discontinuity and compression of the image must be accounted for.

## 8.2 IMAGE AND IMAGE-RELATED DATA STRUCTURES

The linker creates the image structure, including the image header, image section descriptors, symbol vector, and fixup sections. These are interpreted by the Install utility and the image activator, which must understand the data structure versions used by the linker. The Install utility builds the known file database to describe those images that have been identified for special processing, and maintains a list of image control blocks (IMCBs) for preactivated resident images. The image activator initializes an IMCB to describe each image being activated in a process and accesses the known file database. The following section describes the data structures in the image file. Section 8.2.2 discusses IMCBs, and Section 8.2.3, the known file database.

### 8.2.1 Image Header

The image header consists of a number of variable-sized pieces. At the beginning is the fixed portion of the header, which contains some standard information about the image and pointers to the other parts of the header. Figure 8.4 shows the organization of the header and the layout of its fixed part.

The acronym for the fixed part of the header is EIHD. The E distinguishes it from IHD, the acronym for the comparable OpenVMS VAX data structure. Definitions for Alpha and VAX structures must coexist on the same system to support cross-linking. All acronyms for Alpha image header structures and their field names begin with E. The macro $EIHDDEF defines symbolic offsets for the fixed part. Offsets for the other parts are defined by the macros shown in Figure 8.4.

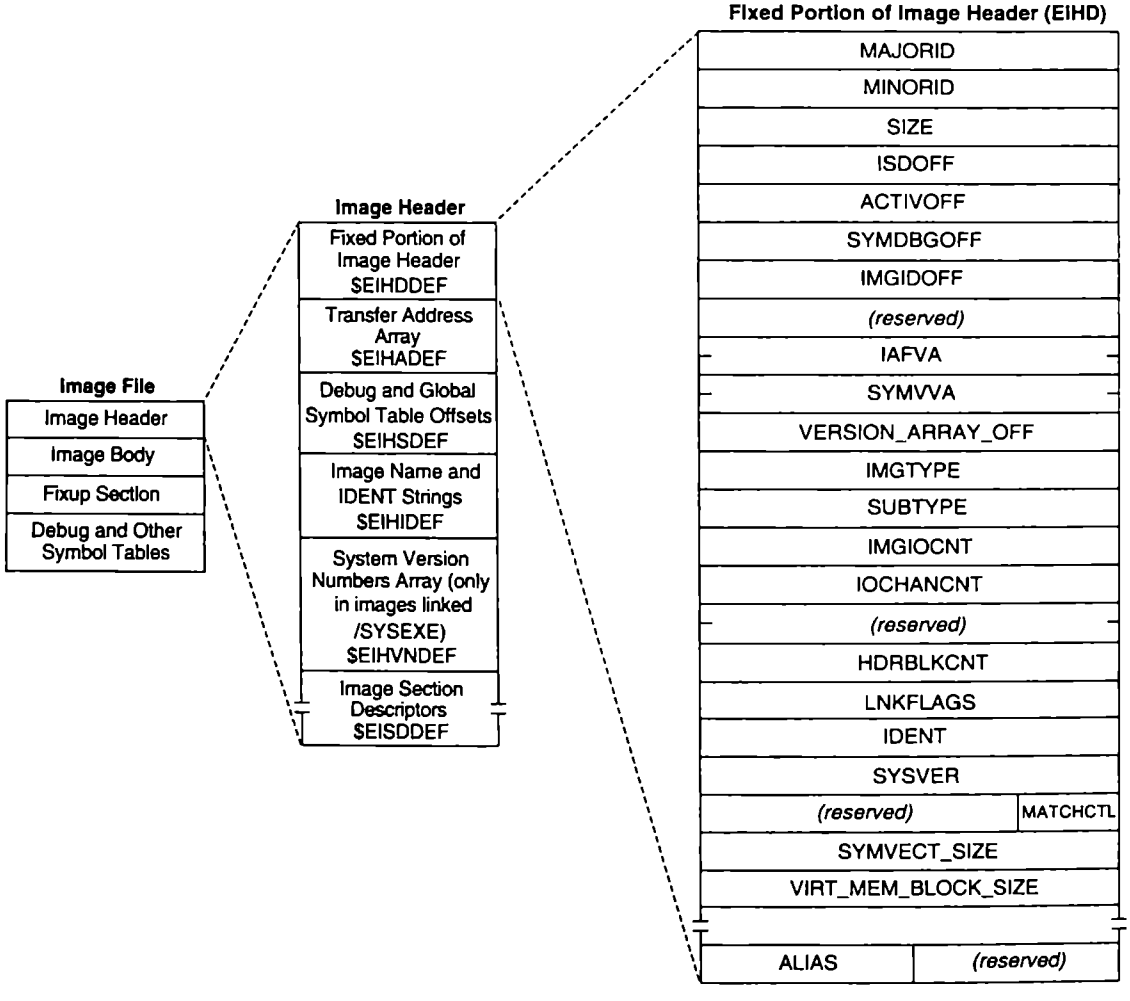



**Figure 8.4**
Contents of an Image and Layout of an Image Header

Section 8.2.1.1 describes the fixed part of the image header. Section 8.5.1 describes the transfer address array portion of the image header. The details of the debug and other symbol tables are beyond the scope of this book. Section 8.2.1.2 describes the image ident area; Chapter *The Modular Executive*, the use of the system version numbers array; and Section 8.2.1.3, image section descriptors.

8.2.1.1 **Fixed Part of the Image Header (EIHD).** EIHD$L_MAJORID and EIHD$L_MINORID contain the major and minor IDs associated with the structure of the image file. For OpenVMS Alpha Versions 1.5 through 7.0, the major ID is 3 and the minor ID is 0.

EIHD$L_SIZE contains the size of the entire image header in bytes, and EIHD$L_HDRBLKCNT, its size in blocks. EIHD$L_ISDOFF, EIHD$L_ACTIVOFF, EIHD$L_SYMDBGOFF, EIHD$L_IMGIDOFF, and EIHD$L_VERSION_ARRAY_OFF contain byte offsets from the start of the image header to the variable-length areas of the image header.

EIHD$Q_IAFVA contains the unrelocated virtual address in the image of its fixup section. For a shareable image, EIHD$Q_SYMVVA contains the unrelocated virtual address in the image of the symbol vector, and EIHD$L_SYMVECT_SIZE, its size in bytes.

EIHD$L_IMGTYPE specifies whether the image is executable (EIHD$K_EXE) or shareable (EIHD$K_LIM, for "linkable image"). EIHD$L_SUBTYPE contains 0 except for an image identified as a CLI through the undocumented linker option IMAGE_TYPE; a CLI has a subtype value of 1.

EIHD$L_IMGIOCNT contains the number of pagelets of image I/O segment requested through the IOSEGMENT option when the image was linked. If the option is not specified, the field contains zero, and the image activator will create a segment whose size is based on the SYSGEN parameter IMGIOCNT.

EIHD$L_LNKFLAGS contains flags that describe the image and the way in which it was linked:

- EIHD$V_LNKDEBUG, when set, means the image has been linked with the qualifier /DEBUG.
- EIHD$V_LNKNOTFR, when set, means the image has no transfer address.
- EIHD$V_NOP0BUFS, when set, means the image was linked with the link option IOSEGMENT=*n*,NOP0BUFS. This value for the option prevents additional image I/O segment pages from being allocated in P0 space. Typically, an image is linked in this way for greater control over the layout and allocation of P0 space.
- EIHD$V_PICIMG, in an OpenVMS VAX image, when set, means the image is position-independent. It has no meaning on OpenVMS Alpha systems.

- EIHD$V_POIMAGE, when set, means the image has been linked with the qualifier /POIMAGE to ensure that its stack and image I/O segment will be allocated in P0 space. Typically, an image is linked in this way to enable it to allocate P1 space for uses that outlive its activation.
- EIHD$V_DBGDMT, when set, means the image's header contains a debug module table for possible use by traceback or a debugger.
- EIHD$V_INISHR, when set, indicates a shareable image with an initialization routine.
- EIHD$V_XLATED, when set, means the image was translated from an OpenVMS VAX image by the VAX Environment Software Translator (VEST) utility.
- EIHD$V_BIND_CODE_SEC, when set, means that the image has been linked in such a way that its code sections can be installed resident in system space.
- EIHD$V_BIND_DATA_SEC, when set, means its nonshared data sections contain no dependencies on image section layout, and so can be compressed.

For an executable image, EIHD$L_IDENT and EIHD$B_MATCHCTL are not meaningful. For a shareable image, EIHD$L_IDENT contains the global section major and minor IDs specified with the GSMATCH link option. EIHD$B_MATCHCTL, also specified with the GSMATCH link option, contains the match algorithm used to determine whether an image linked with a different version of the shareable image can be activated with this version, or whether it must be relinked. It is used in conjunction with the major and minor IDs of the shareable image.

For an image linked with the qualifier /SYSEXE, EIHD$L_SYSVER contains the constant SYS$K_VERSION, the system version number associated with SYS$BASE_IMAGE at the time of the link. Otherwise, it contains zero.

EIHD$L_VIRT_MEM_BLOCK_SIZE represents the grain size for image sections in this image. Its value is a power of 2 based on the /BPAGE qualifier with which the image was linked. Each image section begins on a boundary that is a multiple of the grain size.

EIHD$W_ALIAS is reserved for support of the DECnet Maintenance Operations Protocol, which enables systems to boot from a network device.

8.2.1.2 **Image Ident Area of the Image Header (EIHI).** Figure 8.5 shows the layout of the image ident area of the header. EIHI$L_MAJORID and EIHI$L_MINORID contain the major and minor IDs associated with the structure of this section of the header. For OpenVMS Alpha Version 7.0, the major ID is 1 and the minor ID is 2. EIHI$Q_LINKTIME contains the system time at which the image was linked.

EIHI$T_IMGNAM is a counted ASCII string containing the name of the image. The linker takes its value from the first of the following that specifies a name:

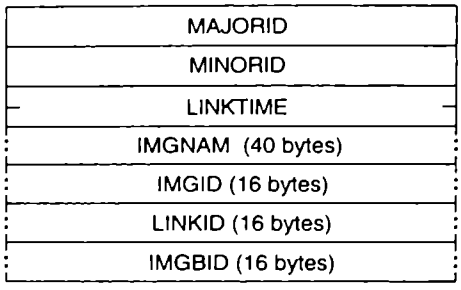



**Figure 8.5**
Layout of Image Ident Area (EIHI)

- The link option NAME, if present
- The name specified with the /EXE or /SHARE qualifier
- The name of the first object module on the link command line

If none specifies a name, the name is blank (the name length is zero).

EIHI$T_IMGID contains the image identification. In an executable image the identification comes from the ID of an object module containing a transfer address. The link option IDENTIFICATION overrides the default. A shareable image does not normally have a transfer address, so either the ID of the first object module or the IDENTIFICATION value is used. EIHI$T_LINKID contains the linker's own identification. Its contents are a function of the version of the linker that built the image. EIHI$T_IMGBID is determined by the linker option BUILD_IDENT and identifies the build class of the image, for instance, a Code Management System (CMS) class detailing the version of all modules used. OpenVMS Alpha Version 7.0 images contain a build class of X65G-SSB-0000.

8.2.1.3 **Image Section Descriptors (EISDs).** The image header contains one EISD for each section in the image that describes a portion of the image and its location in an image file and in virtual address space. The image header also contains a global EISD identifying each shareable image linked with this image. Figure 8.6 shows the layout of the EISD.

EISD$L_MAJORID and EISD$L_MINORID contain the major and minor IDs associated with the image section structure. For OpenVMS Alpha Version 7.0, the major and minor IDs are both 1. The size of the EISD structure itself is stored in EISD$L_EISDSIZE, and EISD$L_SECSIZE contains the number of bytes of memory that the image section requires. In an executable image, EISD$B_TYPE is set to either EISD$K_NORMAL or EISD$K_USRSTACK. In a shareable image, EISD$B_TYPE indicates whether the image section is private or shareable.

The image activator processes EISDs based largely on flags in the field EISD$L_FLAGS and, to a lesser extent, on the contents of EISD$B_TYPE:

- Demand zero EISD. Identified by the flag EISD$V_DZRO, a demand zero EISD describes a range of virtual addresses that begin as zero-filled pages.

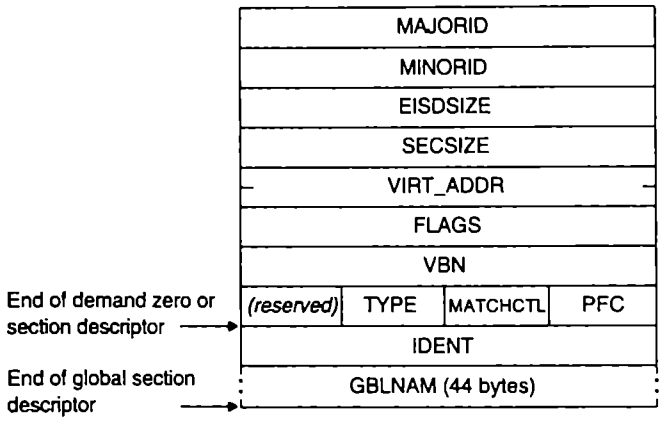



**Image Section Flags**

| Bit | Meaning |
|---|---|
| 0 | Global |
| 1 | Copy-on-reference |
| 2 | Demand zero |
| 3 | Writable |
| 4 | Initialization section |
| 5 | Based section |
| 6 | Fixup section |
| 7 | Installed memory-resident |
| 8 | Vector in image section |
| 9 | Protected image section |
| 10 | Last cluster |
| 11 | Code image section |
| 12 | Non-shareable address data |

**Figure 8.6**
Layout of an Image Section Descriptor (EISD)

The range will be mapped beginning at the virtual address in EISD$Q_VIRT_ADDR unless field EISD$B_TYPE identifies the EISD as the one that describes the user access mode stack. The user mode stack for the initial kernel thread is created at the low-address end of currently allocated P1 space. EISD$L_SECSIZE contains the length of the demand zero section in bytes.

- EISD for a section. A section EISD describes a range of virtual addresses initially filled with code or data from the image file. The image section begins in the image file at the virtual block number in the field EISD$L_VBN; its length in bytes is in EISD$L_SECSIZE.

  EISD$B_PFC, the pagelet fault cluster factor for the section, will be converted to pages and propagated to the process section table entry built to describe the section in the activated image. Its value is based on the pagelet parameter to the linker CLUSTER option and defaults to the SYSGEN parameter PFCDEFAULT. The image section will be faulted into virtual address space beginning at the virtual address in EISD$Q_VIRT_ADDR.

- Global EISD. Marked by the flag EISD$V_GBL, a global EISD identifies a shareable image linked with the current image. The shareable image's name followed by the string _001 is stored as a counted string in the field EISD$T_GBLNAM. The name is based on one of the following:
  - The file name part of the file specification supplied on the link option line that includes the shareable image
  - The module name for a shareable image taken from a shareable image library

  The string _001 is appended to the name to match the algorithm formerly used by the Install utility to create global section names from an image installed /SHARED. However, this portion of the field is ignored.

EISD$L_IDENT and EISD$B_MATCHCTL are taken from the version identification and match control of the shareable image described by the EISD. EISD$Q_VIRT_ADDR contains zero.

A main image linked without any shareable images contains only the first two types of EISD.

A main image linked with a shareable image contains the shareable image's first EISD, primarily to identify the shareable image. The shareable image contains its own image header and EISDs to describe its virtual address space. Address space for the shareable image is not assigned when the main image is linked but when the shareable image is activated. Thus, the size of the shareable image can change without requiring the main image to be relinked.

A shareable image linked with a second shareable image contains a global EISD to identify the second shareable image. If a main image refers only to symbols in the first shareable image but not the second, it does not contain a global EISD for the second shareable image. In this regard, Alpha images differ from VAX images. The entire collection of shareable images implied by a main image is not determined until image activation. Thus, a shareable image can be relinked to reference other shareable images without requiring the relink of the main image linked with it.

The use of these types of EISD is discussed in more detail in later sections.

### 8.2.2 Image Control Block

Figure 8.7 shows the layout of an IMCB. The Install utility allocates IMCBs from paged pool for images installed with shareable address data. It links these IMCBs from the systemwide listhead IMG$GL_IMCB_LIST. The image activator initially allocates IMCBs from the P1 allocation region (see Chapter *Pool Management*) but deallocates them to an IMCB lookaside list for faster subsequent allocation.

The image activator keeps two queues of IMCBs for each process—one for images already activated and one for images yet to be activated. The listheads for these queues begin at the following P1 space locations:

- IAC$GL_IMAGE_LIST—Activated images (known as the done list)
- IAC$GL_WORK_LIST—Images to be activated (known as the work list)
- IAC$GL_ICBFL—Lookaside list of IMCBs

IMCB$B_ACT_CODE describes the manner in which the image was activated—as a main image, a merged image, or a shareable image section. IMCB$B_ACCESS_MODE contains the access mode specified in the $IMGACT request, maximized with the requestor's access mode. That is, IMCB$B_ACCESS_MODE contains the least privileged of these two access modes. The image file is opened on a channel assigned in this access mode,

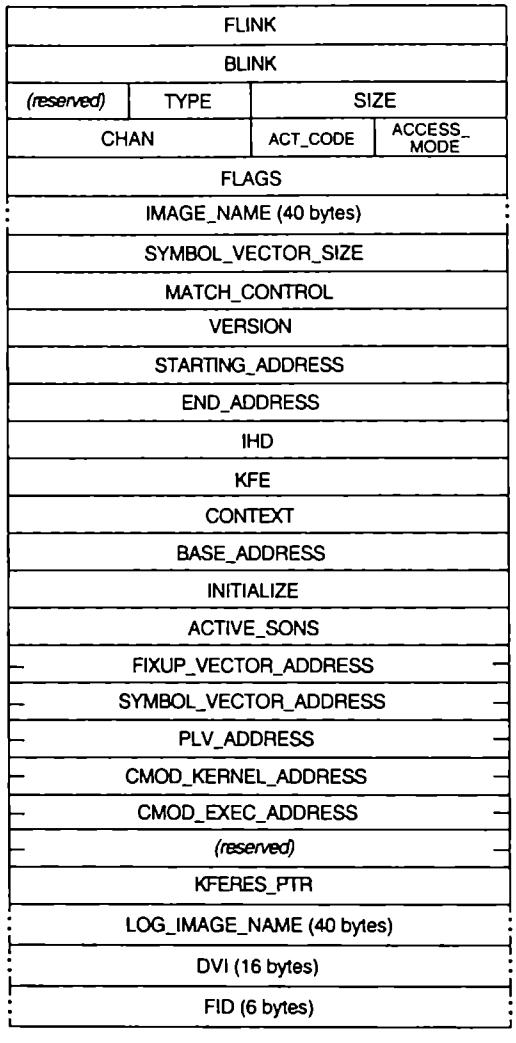



**Image Flags**

| Bit | Meaning |
|---|---|
| 0 | Mapped at end of address space |
| 1 | Installed /SHARED |
| 2 | Has writable sections |
| 3 | Header already decoded |
| 4 | Load from sequential device |
| 5 | Contains initialization code |
| 6 | Completely activated |
| 7 | Linked /SYSEXE |
| 8 | Involved in circularity |
| 9 | Mapped in address space |
| 10 | Installed /PROTECTED |
| 11 | Parent image installed /PROTECTED |
| 12 | Change mode transfer routines mapped |
| 13 | Translated VAX image |
| 14 | Contains protected sections |
| 15 | Contains no protected sections |
| 16 | Installed /RESIDENT |
| 17 | Allows POSIX fork |
| 18 | Data sections compressed |
| 19 | Exempt from system version check |
| 20 | Primary fixups already done |
| 21 | Read-only data in granularity hint region |
| 22 | Installed with shareable linkage |
| 23 | Automatically activated |

**Figure 8.7**
Layout of an Image Control Block (IMCB)

and the pages that are mapped are owned by this mode. IMCB$W_CHAN holds the channel number on which the image file is opened.

The image's fully resolved file name is stored as a counted string in the field IMCB$T_IMAGE_NAME. Saving the complete name rather than a logical name prevents problems when a shareable image is referenced by both a native and a translated image. Under these circumstances, multiple activations of the same image would otherwise be possible. (Chapter 9 contains more information on the mechanisms that enable native and translated images to interoperate.)

IMCB$T_LOG_IMAGE_NAME contains the old-format global section name of a shareable image's first global section.

The address range into which the image was mapped is stored in IMCB$L_STARTING_ADDRESS and IMCB$L_END_ADDRESS. For an image in-

stalled /RESIDENT, these addresses represent only the sections mapped in process-private address space, either P0 space or, for an image installed /RESIDENT/SHARE=ADDRESS_DATA, in P1 space.

IMCB$L_IHD points to the image header if the image is installed /HEADER_RESIDENT. IMCB$L_KFE locates the known file entry associated with the image (if any), and IMCB$L_CONTEXT points to the image activator local context block, a temporary structure that points to image activator buffers. IMCB$PS_FIXUP_VECTOR_ADDRESS contains the address of the fixup section in the image.

For a shareable image, IMCB$PS_SYMBOL_VECTOR_ADDRESS and IMCB$L_SYMBOL_VECTOR_SIZE contain the address and size in bytes of the image's symbol vector. IMCB$L_INITIALIZE contains the unrelocated procedure value of the image's initialization routine, taken from EIHA$Q_INISHR (see Section 8.5.1).

For a privileged shareable image, IMCB$PS_PLV_ADDRESS contains the address of the image's privileged library vector. IMCB$PS_CMOD_KERNEL_ADDRESS and IMCB$PS_CMOD_EXEC_ADDRESS contain the beginning addresses of the sets of kernel and executive mode transfer routines created by the image activator.

For an image installed /RESIDENT, IMCB$L_KFERES_PTR contains the address of a user-readable P1 space copy of the system space data structure that describes the resident sections.

For images whose file headers are not already in memory, for example, uninstalled images or those not installed /HEADER_RESIDENT, IMCB$T_DVI and IMCB$W_FID contain the device and file identifiers of the image file.

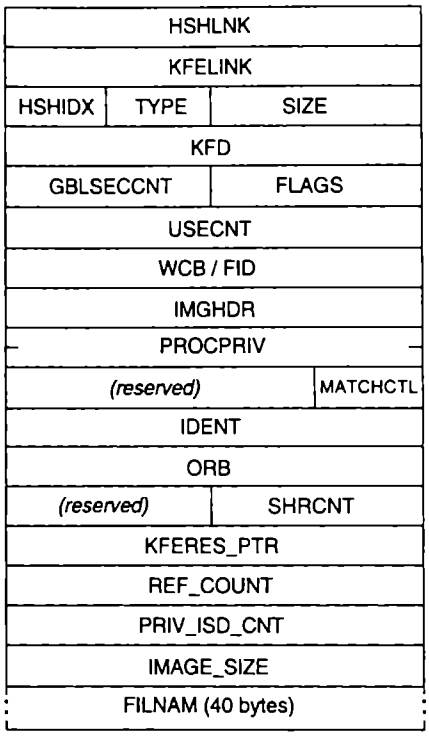
### 8.2.3 Known File Database

A number of data structures describe known images:

- Known file entry (KFE)—one for each known image
- Known file resident image header (KFRH)—one for each image installed /HEADER_RESIDENT
- Known file directory (KFD)—one for each unique combination of mass storage device and directory from which known images are installed
- Resident section descriptor (KFERES)—one for each image installed /RESIDENT or /RESIDENT/SHARE=ADDRESS_DATA
- One hash table to locate all KFEs
- One known file pointer block (KFPB) to locate the KFD list and KFE hash table

These data structures are all allocated from paged pool.

**8.2.3.1 Known File Entry.** The Install utility allocates a KFE when a known image is installed. A KFE contains information used by the image activator to locate and map the image. Figure 8.8 shows the layout of a KFE.




**KFE Flags**

| Bit | Meaning |
|---|---|
| 0 | Installed /PROTECTED |
| 1 | Shareable image |
| 2 | Installed /PRIVILEGED |
| 3 | Installed /OPEN |
| 4 | Image header resident |
| 5 | Shared image |
| 6 | Use KFE-based names for global sections |
| 7 | *(reserved)* |
| 8 | Installed /NOPURGE |
| 9 | Image accounting enabled |
| 10 | Has writable sections |
| 11 | Execute access only |
| 12 | Installed /RESIDENT |
| 13 | Delete pending |
| 14 | Exempt from system version checks |
| 15 | Read-only data in granularity hint region |

**Figure 8.8**
Layout of a Known File Entry (KFE)

KFE$L_FID and KFESL_WCB are different symbolic names for the same location in the KFE. If the image is not installed /OPEN, three words beginning at KFESL_FID contain the full file ID of the image, thus locating the file header on the disk. Otherwise, if the file header is already in memory, KFE$L_WCB contains the address of the file's WCB, which describes the disk location of the blocks of an open file. KFE$L_IMGHDR contains the address of the resident image header.

The field KFE$W_FLAGS contains flag bits indicating the manner in which the image was installed—for example, if KFE$V_PROTECT is set, the image was installed /PROTECTED. The image activator maintains a count of the number of processes that have activated the image at KFE$L_USECNT. An image installed with privileges has its privilege mask recorded in KFE$Q_PROCPRIV. KFE$L_IMAGE_SIZE contains the size in bytes of an installed image.

Several fields are used only for images installed /SHARED, or installed in a way that implies /SHARED, such as /RESIDENT:

- KFE$W_GBLSECCNT is a count of the total number of image section descriptors in the image file. Despite the name, this field is not the count of image sections for which the Install utility created system global sections. That information can only be obtained by walking the systemwide list of global section descriptors.

- KFE$L_PRIV_ISD_CNT contains the number of private image sections, those that are not references to a shareable image linked with this image and that do not describe the stack. This count is the total number of system global sections to be mapped and process-private sections to be created when the image is activated.
- KFE$L_IDENT contains the global section identifier.
- KFE$B_MATCHCTL contains the match control information supplied when the image was linked.

For an image installed /RESIDENT, KFE$L_KFERES_PTR contains the address of the data structure that describes the resident sections and KFE$L_REF_COUNT contains the number of processes currently accessing the image.

8.2.3.2 **Known File Resident Image Header.** A KFRH exists for each known image installed /HEADER_RESIDENT. The KFRH immediately precedes the image header, and space for the image header is allocated with the KFRH. Figure 8.9 shows the layout of a KFRH.

8.2.3.3 **Known File Directory.** Figure 8.10 shows the layout of a KFD. The full device and directory names associated with an installed image are stored in the KFD field KFD$T_DDTSTR rather than in the KFE. Typically, multiple known images are installed from the same device and directory combination and

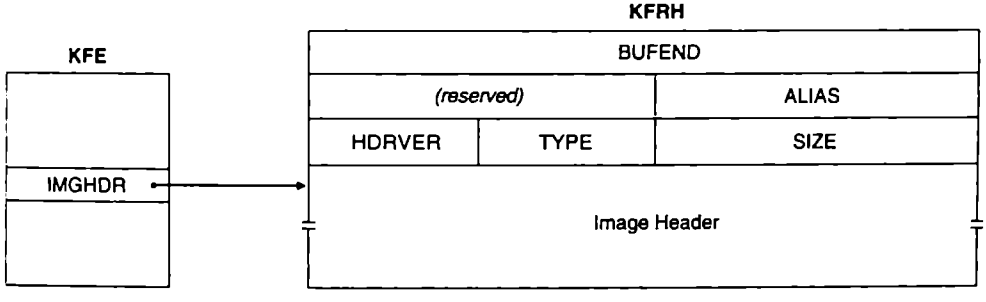



**Figure 8.9**
Layout of a Known File Resident Image Header (KFRH)

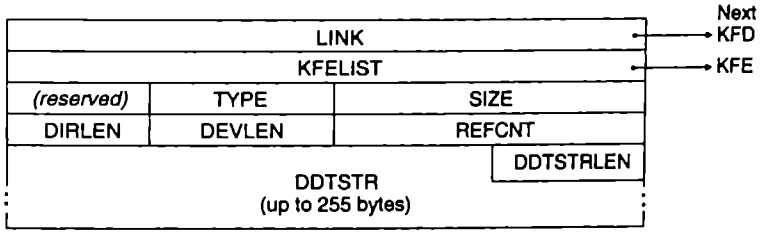



**Figure 8.10**
Layout of a Known File Directory (KFD)

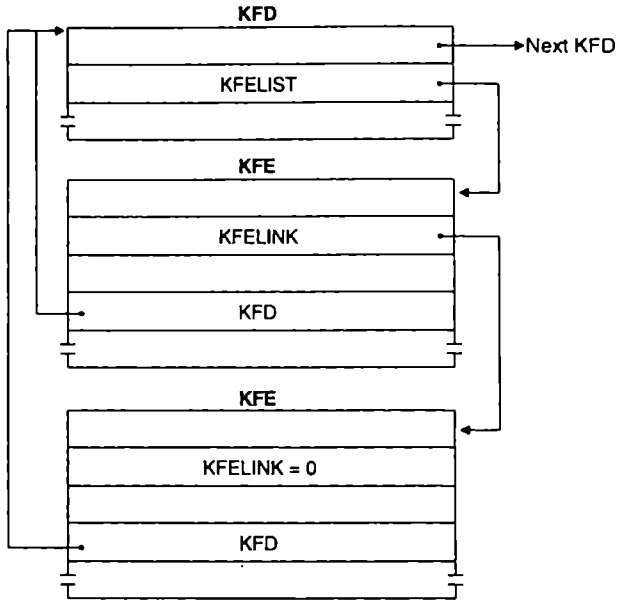



**Figure 8.11**
Known File Directory (KFD) and Known File Entries (KFEs)

thus share the same KFD. Keeping the device and directory information in the KFD rather than in each KFE saves paged pool. The number of KFEs sharing a KFD is found in KFD$W_REFCNT. The KFEs themselves are linked at KFD$L_KFELIST. Figure 8.11 shows a KFD and its list of KFEs.

8.2.3.4 **KFE Hash Table and KFPB.** A KFE hash table locates all the KFEs. A known image name is hashed to a number between 0 and 127, which is an index into the 128-entry hash table. If the table entry contains a zero, no KFE is associated with that hash index. Otherwise, the table entry is the address of a KFE. As a confirmation, the KFE contains its own hash index value at KFE$B_HSHIDX. KFEs with the same hash index are linked together through the field KFE$L_HSHLNK. The end of the list is a forward link of zero. Figure 8.12 shows the hash table and several KFEs linked to it.

The KFPB contains the hash table address at KFPB$L_KFEHSHTAB and the number of hash table entries at KFPB$W_HSHTABLEN. It also holds the head of the KFD list at KFPB$L_KFDLST and the KFD count at KFPB$W_KFDLSTCNT. Figure 8.12 shows the layout of the KFPB and its relation to other known image data structures.

8.2.3.5 **KFE Resident Section Descriptor.** For each image installed /RESIDENT, a KFERES structure is built in paged pool. Figure 8.13 shows the layout of this structure. As such an image is activated, a P1 copy of the structure is built in the process allocation region. The IMCB points to this copy; the KFE

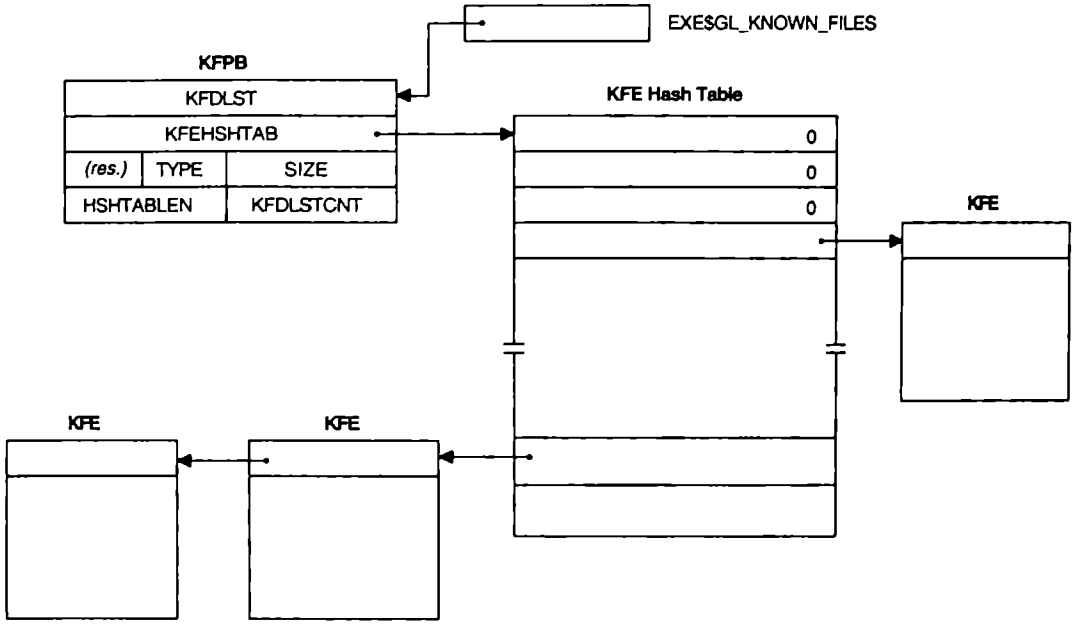



**Figure 8.12**
Layout of a Known File Pointer Block (KFPB) and a KFE Hash Table

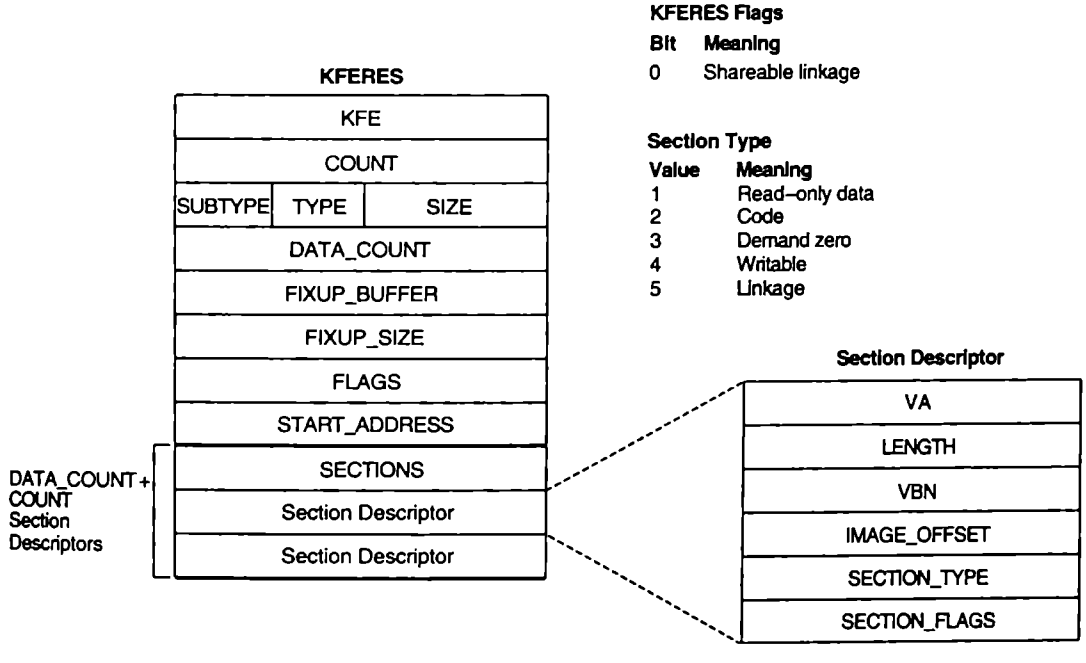



**Figure 8.13**
Layout of a Known File Entry Resident Section Descriptor (KFERES)

points to the original in paged pool. The P1 copy of the structure serves the following purposes:

- User-readable, it can be accessed by the $IMGFIX system service, a mode of caller service.
- Unique to a particular process, it can describe the nonshareable sections of the image, which are allocated in process-private address space.

KFERES$L_KFE contains the address of the KFE that describes the image, and KFERES$W_SIZE, KFERES$B_TYPE, and KFERES$B_SUBTYPE are the standard dynamic data structure fields.

The only bit defined in KFERES$L_FLAGS indicates whether the image is installed with shareable address data. When this flag is set, KFERES$L_FIXUP_BUFFER contains the address of a subset fixup section for fixups targeting process-private sections. This section is created in paged pool by the Install utility and copied to process-private space at image activation; the paged pool KFERES points to the paged pool fixup section, and the P1 copy KFERES points to the process-private copy. KFERES$L_FIXUP_SIZE contains the length of the subset fixup section in bytes. KFERES$L_START_ADDRESS contains the process-private address of the first nonresident image section.

KFERES$L_COUNT contains the number of resident sections in the image, and KFERES$L_DATA_COUNT, the number of other image sections mapped in process-private address space and described in the KFERES. The KFERES contains substructures, called section descriptors, beginning at offset KFERES$T_SECTIONS. Adding the contents of the KFERES$L_COUNT and the KFERES$L_DATA_COUNT fields gives the number of section descriptors present in the KFERES.

For an image installed without shareable address data, the number of section descriptors and the value in KFERES$L_DATA_COUNT can grow when the paged pool KFERES is copied to P1 space during image activation if the image is linked with /SECTION_BINDING=DATA. The paged pool KFERES contains only section descriptors for resident sections; the P1 copy receives additional section descriptors to describe the remaining image sections if data compression is enabled. As each of these sections is allocated in process-private address space, KFERES$L_VA is set to the starting virtual address of each section, and KFERES$L_LENGTH to its length in bytes.

For an image installed with shareable address data, however, the placement of each image section is predetermined by the Install utility. The paged pool KFERES contains one section descriptor for each resident section and one for each image section to be mapped in P1 space. In this case, KFERES$L_VA in each section descriptor contains the starting virtual address in P1 space where the section is to be mapped when the image is activated, and KFERES$L_LENGTH, its length in bytes.

KFERES$L_VBN has the starting virtual block number of the section in the image file, and KFERES$L_IMAGE_OFFSET, the starting address of the section in the image as it was linked. For an image installed with shareable address data, KFERES$L_SECTION_TYPE contains the section characteristics. A clear field is interpreted as a compressed data section, although no such symbol exists. When set, KFERES$V_SHARESECT, in KFERES$L_SECTION_FLAGS, indicates a shareable section.

## 8.3 IMAGE ACTIVATION

When a user initiates image execution, for example, as part of process creation or through a DCL command in an interactive or batch job, the $IMGACT system service is requested.

The image activator calls RMS to open the main image file, which enables the system to perform all its file protection checks. Then the image activator reads the image and requests a memory management system service to map each image section within it. The memory management system services initialize the process-private page tables to map the pages in the image file. (The image activator does not load the image into memory; instead, pages of the image will be read into memory by the OpenVMS paging mechanism as they are referenced during image fixup and execution.) The image activator takes similar steps for shareable images with which the main image was linked. It makes other necessary preparations, such as creating address space for the user stack.

As each IMCB is processed, the image activator relocates intra-image references that require relocation. Before control is transferred to the image, the $IMGFIX system service fixes up inter-image references. These resolutions must be performed at activation time rather than at link time because they depend upon the addresses into which the images are mapped.

Control is transferred to the image as described in Section 8.5.

The next two sections cover the more common types of image activation in detail:

- Activation of a simple main image, one linked with no shareable images. This is an artificial separation from the next case, simply to illustrate the difference in the image activator's actions.
- Activation of an image linked with one or more shareable images. Because most language processors generate calls to library routines implemented as shareable images, this case includes most images.

Certain types of image are made known to the system through the Install utility, to speed up their activation, to associate special properties with them, or both. Section 8.3.6 briefly describes the activation of various types of known image, including resident images, resident images with shareable

address data, and privileged shareable images. It also summarizes considerations for translated images.

Table 8.1 shows the arguments to the $IMGACT system service.

Direct requests to this system service are reserved for the OpenVMS Alpha operating system. Direct requests by users are unsupported. Instead, users can request the image activator indirectly through any CLI command that runs an image and through the Run-Time Library procedure LIB$FIND_IMAGE_SYMBOL.

### 8.3.1 Activation of a Simple Main Image

Most of the common operations performed by the image activator occur during the activation of a simple main image, one linked with no shareable images. This section therefore follows the general flow through the image activator for simple main images, including those installed /HEADER_RESIDENT or /SHARED. Note that the concept of a main image linked with no shareable images is a simplification, since any image that requests system services is linked with SYS$PUBLIC_VECTORS.EXE to resolve its system service names.

The $IMGACT system service procedure, EXE$IMGACT, runs primarily in executive mode with some kernel mode subroutines. EXE$IMGACT is in module SYSIMGACT; some of the procedures it calls are in modules IMGMAPISD, IMGDECODE, SYSIMGFIX, and IMGACT_SUPPORT. EXE$IMGACT and the procedures it calls are known as the image activator.

To activate a simple main image, the image activator takes the following steps:

1. It initializes its P1 space scratch area and various P1 cells.
2. It checks the accessibility of the $IMGACT system service argument list and its arguments and copies them for later use.
3. It resets the activated privileged library dispatch vector (see Chapter *System Service Dispatching*) to indicate no activated user-written system services or rundown routines.
4. It calls RM$RESET, in module RMSRESET, to initialize the image I/O segment.
5. It allocates and zeros an IMCB. If an error occurs later in activation, it deallocates the IMCB before returning.
6. It locks the known file database by requesting the Enqueue Lock Request and Wait ($ENQW) system service. It locks the systemwide resource INSTALL$KNOWN FILE for protected read. This blocks any attempt at concurrent changes to the known file database by the Install utility. If an error occurs later in activation, the image activator dequeues the lock before returning.

Table 8.1 Arguments to the $IMGACT System Service

| *Argument Name* | *Meaning* |
|---|---|
| NAME | Descriptor of image name to be activated. |
| DFLNAM | Descriptor of default file name. |
| HDRBUF | Address of 512-byte buffer in which the EIHD and image file descriptor are returned. The first two longwords in the buffer are the addresses within the buffer of the EIHD and the image file descriptor. |
| IMGCTL | Image activation control flags: |
| *Flag* | *Meaning* |
| IAC$V_MERGE | If set, the image activator is directed to merge an image into the address space of an already activated image, ignoring the user stack and the image I/O segment. This flag must be set if the service is requested from user mode. |
| IAC$V_EXPREG | If set, the INADR argument does not give an actual address range but merely indicates P0 address space, which is expanded as required. This flag is used only during a merged image activation for a P0 image. |
| IAC$V_SETVECTOR | If set, the image activator initializes only the P1 vectors that dispatch to user-written system services, rundown routines, and message sections. |
| IAC$V_P1MERGE | If set, the image activator is directed to merge an executable image into P1 space. It first merges it into P0 space to determine its size and thus the correct starting address in P1 space and then maps it into P1 space. |
| IAC$V_P1DIRECT | If set, the image activator activates an image directly into P1 space. This flag is intended solely for DCL.EXE's activation. |
| IAC$V_PROTECTED | If set, the image activator requires the image and any activated with it to have been installed /PROTECTED. Logical name translations use inner mode logical names and tables. |
| IAC$V_PARANOID | If set, the image activator requires the image and any activated with it to have been installed. Logical name translations use inner mode logical names and tables. |
| IAC$V_DEBUG | If set, the debugger is activated with the image. |
| IAC$V_PERMANENT | If set, the image activator makes image vectors permanent. Used for resident images installed with shareable address data. |
| IAC$V_NOTRANS | If set, the image activator skips logical name translation when opening the image file. Used for translated images. |
| INADR | Address of a two-longword array containing the virtual address range into which the image is to be mapped. This argument is usually omitted, in which case the address ranges designated by the EISDs in the EIHD are used or the image is mapped at the next available location. |
| RETADR | Address of a two-longword array to receive the starting and ending addresses into which the image was actually mapped. |
| IDENT | Address of a quadword containing the version number and matching criteria for a shareable image. |
| ACMODE | Access mode for page ownership and image channel assignment. It defaults to user mode. If present, it is maximized with the requestor's mode. |

7. The image activator requests RMS to open the image for execute access, specifying the user-open, process-permanent file, sequential-only, and known file database search options. It specifies that the WCB is to contain complete mapping information for the file, thus avoiding later window turns.

   When activating any of the following image types, the image activator specifies that RMS use only executive or kernel mode logical names to translate the image name and its descendants' image names:

   - A main image installed /PRIVILEGED
   - A main image installed /EXECUTE_ONLY and activated from user mode
   - A main image invoked by a process with execute access but not read access to the image file
   - An image installed /PROTECTED or having an ancestor installed /PROTECTED
   - An image activated with the IAC$V_PARANOID flag specified in the FLAGS argument

8. The image activator then stores the file ID, device name, translated image name, and channel number in the IMCB.
9. If RMS found the image in the known file database, it returned the address of the KFE in the CTX field of the file access block (FAB); the image activator stores the KFE address in the IMCB and notes whether the image was installed with the /PRIVILEGED, /ACCOUNT, /PROTECTED, /EXECUTE_ONLY, or /SHARED qualifiers. If KFE$V_KFE_NAMING is set, the image activator initializes its internal global section name descriptor with the address of the KFE. In this case, the default, it will base global section names on KFE address and EISD number. Earlier versions of OpenVMS based global section names on the image file name.
10. If either IAC$V_PARANOID or IAC$V_PROTECTED was specified in the FLAGS argument, the image activator checks that the image was installed and, if not, returns the error status SS$_PRIVINSTALL to its requestor. If IAC$V_PROTECTED was specified, the image must also have been installed /PROTECTED. If the image passes these checks, the image activator sets IMCB$V_PARENT_PROT, which is propagated to shareable images activated subsequently to ensure that similar checks are made for them.
11. The image activator tests whether the image header is resident. A known image with its header resident in memory can be activated quickly because a header read operation is avoided. If the header is resident, the image activator sets IMCB$V_RES_HEADER, copies the symbol vector address and size from the image header to the IMCB, and records in the IMCB whether the image was installed with resident code, data, or com-

pressed data sections. It then skips to step 14 for images with resident headers.

12. Otherwise, when the image header is not resident, routine IMG$DECODE_IHD, in module IMGDECODE, reads the first block of the image file and examines the first longword in that block to determine if the image is an OpenVMS Alpha image. If the value in the longword is out of range, the image activator returns the error status IMG$_NOTNATIVE to its requestor. If the value is within range, it performs several consistency checks to determine the validity of the header and image. It also copies the symbol vector address and size to the IMCB.
13. The image activator checks that the format of the image header itself matches that known to the image activator by validating the image header major and minor ID values. If there is a mismatch, the image activator aborts the activation and returns the error status SS$_BADIMGHDR.
14. If the image being activated has a KFE, the image activator tests KFE$V_VERSION_SAFE and copies this flag to the IMCB if set. KFE$V_VERSION_SAFE is set by the Install utility when installing the image if it discovers a record in the system version override file, SYS$SYSTEM:VMS$IMAGE_VERSION.DAT, that matches the image's name, image ID, and link time. Such a record indicates that the image's use of internal system interfaces is certified as safe under the current OpenVMS release.

    Otherwise the image activator calls EXE$CHECK_VERSION, in module CHECK_VERSION, which checks whether an image linked against SYS$BASE_IMAGE.EXE is compatible with the versions of those symbols in the running system. Chapter *The Modular Executive* describes the compatibility check in detail. If the versions are incompatible, the image activator performs its own check of the system version override file. Failure causes it to abort the activation and return the fatal error status SS$_SYSVERDIF.
15. The image activator tests whether the image is an ordinary OpenVMS Alpha image or a CLI by examining the contents of EIHD$L_SUBTYPE. If the field contains EIHD$C_CLI, then the image is a CLI and the image activator instead activates LOGINOUT. Section 8.3.6 contains further details about the activation of a CLI.
16. The image activator copies information from the $IMGACT system service argument list into the IMCB and inserts the IMCB at the tail of its work list, which was previously empty. It determines such things as IMCB$L_STARTING_ADDRESS and IMCB$L_END_ADDRESS. It stores the constant IMCB$K_MAIN_PROGRAM in IMCB$B_ACT_CODE to indicate that this is the main image.
17. The image activator enters its main loop, IMG$DO_WORK_LIST, in module IMGMAPISD. It begins processing the work list by removing

an IMCB from the head of the list. The first IMCB removed from the work list is the IMCB describing the main image, which was inserted in step 16.

18. The image activator processes the EISDs in the image header. Its main task is setting up the process-private page tables to reflect the address space produced by the linker. It reads each EISD in the image header (see Figure 8.6) and determines the type of section described: private or demand zero for the simple main image in this example; private, demand zero, or global for a main image linked with shareable images. It then requests the appropriate memory management system service to perform the actual mapping.

    - The most common form of EISD describes a private section. A private section is either read-only or read/write, depending on the attributes of the PSECTs that compose the image section. Initial page faults for all pages in a private section are satisfied from the appropriate blocks in the image file.

      To map a private section into process address space, the image activator normally requests the Create and Map Section ($CRMPSC) system service, using the contents of the EISD as input arguments. It always specifies the NO_OVERMAP flag, so that if pages exist in the desired virtual address range, they are not deleted. If the section is a protected image section or if IACSV_PARANOID was set in the image activator FLAGS argument and the section is a fixup section, it also specifies the $CRMPSC PROTECT flag and a section owner of executive access mode.

      The result is a process section table entry (PSTE) and a series of level 3 page table entries (L3PTEs) containing process section table indexes. If a new L3PT is created, a level 2 PTE is initialized to map it.

      Figure 8.14 shows the relations among the L3PTEs, the PSTE, and the EISD. The shaded bits in the L3PTE are either reserved or bits whose contents are irrelevant for that form of PTE. (Because the section in this example is a P0 space section, the L3PTEs are P0 space L3PTEs, also known as P0 page table entries, P0PTEs.) The number of L3PTEs is equal to the section's size in bytes rounded up to an integral number of pages, divided by page size. The first L3PTE maps the page whose virtual address corresponds to the section's base virtual address. All the L3PTEs index the same process section.
    - When an image is installed /SHARED or in some manner that implies /SHARED, like /RESIDENT, the Install utility processes its EISDs and creates global sections wherever image section characteristics allow. When a process activates such an image, the image activator maps

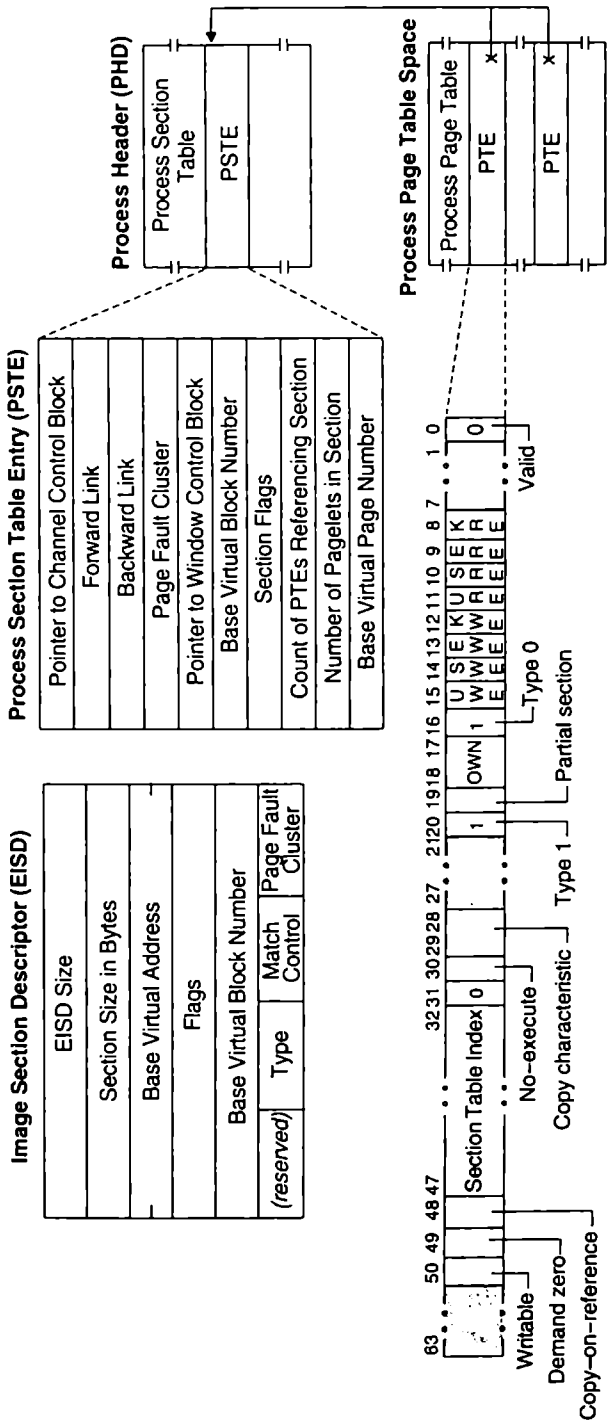



**Figure 8.14**
EISD and Page Table Entries for Process-Private Section

those existing global sections into process address space using the Map Global Section ($MGBLSC) system service and creates private sections for those EISDs that describe nonshareable sections.

If the section is read-only and the image was installed /SHARED, the image activator requests the $MGBLSC system service. The result is a series of L3PTEs that contain consecutive global page table indexes. Figure 8.15 shows the L3PTEs, global page table, and EISD. The shaded bits in the L3PTE are either reserved or bits whose contents are irrelevant for that form of PTE. The number of L3PTEs is equal to the section's size in bytes rounded up to an integral number of pages, divided by page size. The first L3PTE maps the page whose virtual address corresponds to the base P0 virtual address at which the global section is mapped.

If the section is writable and the image was installed /SHARED and /WRITE, the image activator requests the $MGBLSC system service.

If the section is writable, copy-on-reference, and not installed /SHARED, the image activator requests the $CRMPSC system service to create a private copy of the section.

If the section is writable, not copy-on-reference, and not installed /SHARED, the image activator returns the error status SS$_NOTINSTALL to its requestor.

If the section is read-only but not installed /SHARED, it requests the $CRMPSC system service.

One special kind of private section is a fixup section. When the image activator encounters an EISD describing a fixup section, it stores the address of the fixup vector into IMCB$PS_FIXUP_VECTOR_ADDRESS and the address of the IMCB into EIAF$L_BASE_VA in the fixup section. It links the fixup section through EIAF$L_FIXUPLNK to the list of fixup sections to be processed later by the $IMGFIX system service (see Section 8.4).

- Another form of EISD is a demand zero section. The linker produces such a section when uninitialized copy-on-reference pages are requested in the image. The image file on disk does not contain demand zero section pages, just an indication in the EISD that a certain range of virtual address space is uninitialized.

  The image activator uses the contents of this type of EISD as input arguments to an internal interface to the Create Virtual Address Space ($CRETVA) system service. The $CRETVA system service creates new demand zero pages in the specified range of virtual addresses. By default, if it discovers any pages that already exist in the range, they are deleted. The internal interface allows the image activator to specify the NO_OVERMAP flag, overriding this default. The result is a series

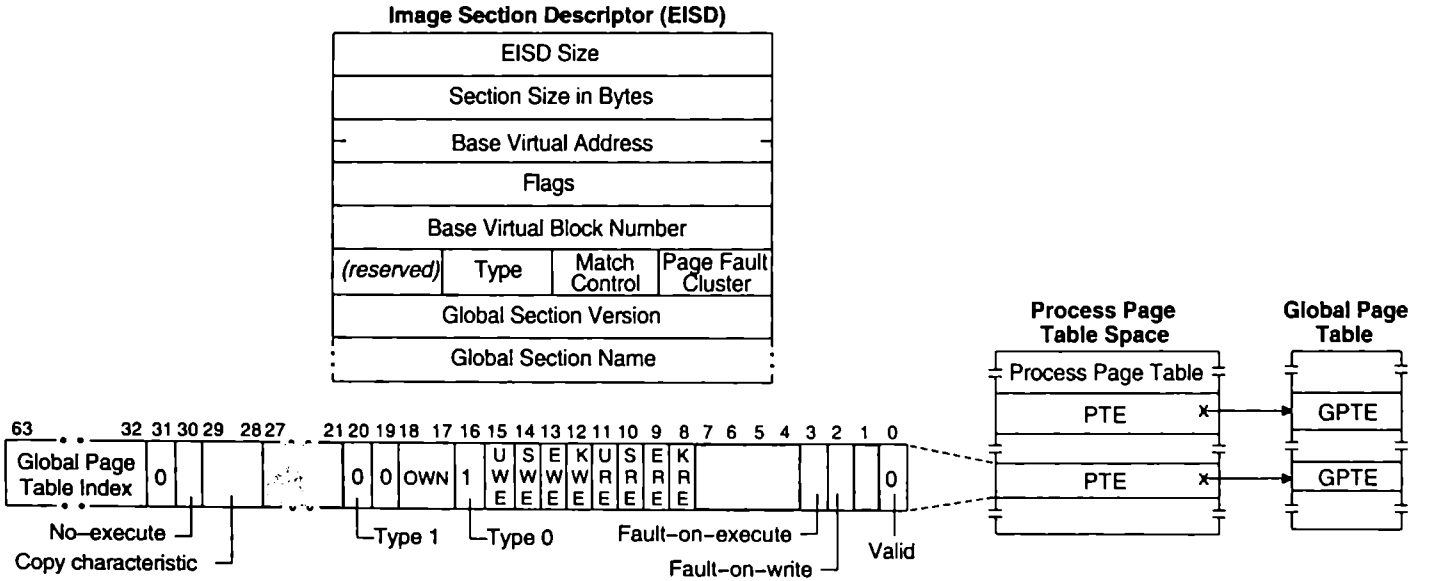



**Figure 8.15**
EISD and Page Table Entries for Global Section

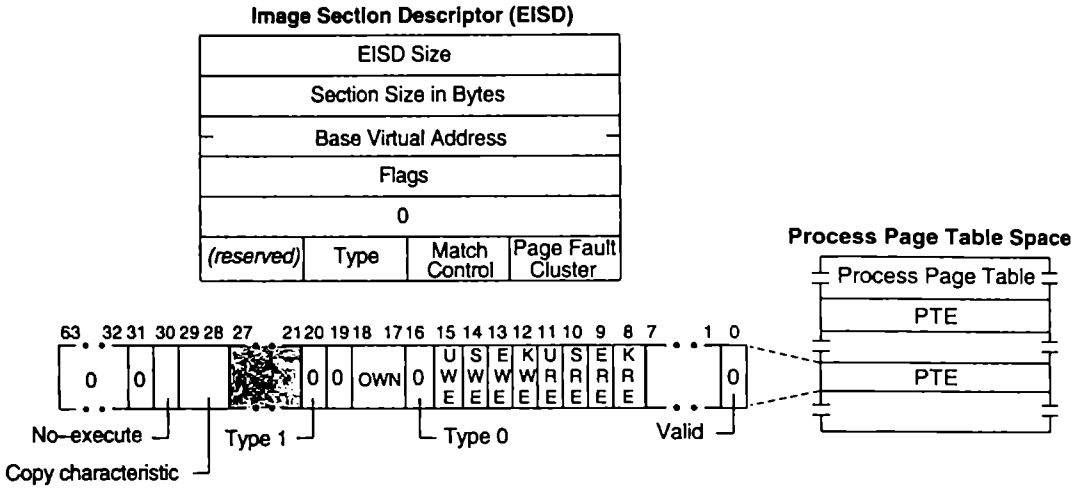



**Figure 8.16**
EISD and Page Table Entries for Demand Zero Section

of demand zero page L3PTEs. The number of L3PTEs is equal to the section's size in bytes rounded up to the next page boundary, divided by page size. Figure 8.16 shows the EISD and PTEs for a demand zero section.

Note that one such section is the area in P1 space that contains the user stack. The linker distinguishes this special demand zero section from others by a special code byte in the type designator in the EISD. The image activator copies the size from the EISD and delays mapping the user stack until later in the activation.

- The third type of EISD, which would not be found in the simple main image of this example, is a global EISD. A global EISD indicates that the image activator must activate a shareable image. When it encounters a global EISD, it builds an IMCB to describe the shareable image, copying match control, version information, and global section name from the EISD. It inserts the IMCB at the tail of its work list. Section 8.3.2 describes the activation of a shareable image.

19. In this example of a simple main image (with no references to shareable images and thus no global EISDs), the only IMCB on the work list has now been processed. The image activator continues with its end processing, described in Section 8.3.3.

    In the case of an image linked with shareable images, the image activator would have found global EISDs while processing the main image IMCB. Thus, additional IMCBs would have been added to the work list. The image activator would process them as described in the following section.

### 8.3.2 Activation of Shareable Images

Image activation is iterative. When the image activator activates an image linked with shareable images, it potentially adds an IMCB for each shareable image to the tail of its work list. As it activates each of those shareable images, it may find more global EISDs in the shareable image's header and create more work list IMCBs.

Commonly referenced shareable images, such as LIBRTL, can appear in the work list multiple times. Activating an image linked with several shareable images, each linked with LIBRTL, causes multiple insertions of LIBRTL into the work list. No matter how many times a shareable image appears in the work list, it is activated only once. As it activates an image, the image activator moves the image's IMCB from the work list to the done list. Once the IMCB is on the done list, the image activator adds no further IMCBs for that image to the work list.

This section describes the activation of a shareable image. The activation could be the result of activating a main image linked with shareable images. Alternatively, the activation could be a merged image activation, for example, one requested through LIB$FIND_IMAGE_SYMBOL, after the main image and its shareable images have all been activated.

The image activator takes the following steps to activate a shareable image:

1. In routine IMG$DO_WORK_LIST the image activator attempts to remove an IMCB from the head of its work list. If there is none, activation is complete and the image activator proceeds with its end processing, described in Section 8.3.3.
2. It checks the done list for an IMCB denoting the same image as the one in the IMCB removed from the work list. The comparison is made on the basis of the fields IMCB$T_DVI and IMCB$T_FID, if they are nonzero. Otherwise, the comparison uses the image name or logical name fields. The existence of such an IMCB indicates that the image named in the work list IMCB has already been activated.

   If there is such an image, the image activator must still ensure that the earlier activation matches current protection requirements. If an image is installed /PROTECTED, all shareable images with which it was linked must be installed. (If several shareable images were linked with the same shareable image X, and only one of those shareable images is installed /PROTECTED, image X might possibly be activated before the /PROTECTED image, that is, before the image activator detects that image X must be an installed image.) The image activator checks for this condition and returns the error status SS$_PRIVINSTALL if the image is not installed. Otherwise, it deallocates the IMCB and goes back to step 1 to process the next IMCB on the work list.

3. If the image has not already been activated, the image activator inserts the IMCB at the top of a stack pointer maintained in the done list. This mechanism ensures that IMCBs appear in the list in the proper order for image initialization (see Section 8.3.4). If an error occurs later in the activation, the image activator removes the IMCB from the done list.
4. If the image is one of the system base images (SYS$BASE_IMAGE.EXE or SYS$PUBLIC_VECTORS.EXE), the image activator copies the starting and ending system space addresses of the corresponding symbol vector to the IMCB. Otherwise, the image activator requests RMS to open the image named by the IMCB. It specifies a default file type of EXE and directory of SYS$SHARE, with file open options of user-open, process-permanent file, sequential-only, and known file database search. The image activator requests a WCB containing complete mapping information for the file, thus avoiding later window turns. If the global EISD specified a writable global section, the image activator requests shared write access. Otherwise, it requests execute access.
5. To locate the file, RMS attempts logical name translation of the file name part of the image name. Under the circumstances described in step 7 of Section 8.3.1, it uses only executive or kernel mode logical names to translate the image name. The image activator makes the checks in step 10 of Section 8.3.1 and confirms that the image returned by RMS is a known image. If not, the activation is aborted and the image activator returns the error status SS$_PRIVINSTALL.

   In addition, if a shareable image is not installed /EXECUTE_ONLY, and the process does not have both read and execute access to it, the activation is aborted and the image activator returns the error status SS$_ACCONFLICT.
6. If the image is a known image with its header resident, the image activator sets IMCB$V_RES_HEADER, copies the symbol vector address and size from the image header to the IMCB, and records in the IMCB whether the image was installed with resident code, data, or compressed data sections. It continues processing in step 9.
7. Otherwise, the image activator reads in the image's header (see step 12 in Section 8.3.1) and copies the symbol vector size and address from the image header to the IMCB.
8. The image activator checks that the format of the image header itself matches that known to the image activator by validating the image header major and minor ID values. If there is a mismatch, the image activator aborts the activation and returns the error status SS$_BADIMGHDR.
9. It verifies that the match control information in the image header is consistent with the match requested in the global EISD whose presence caused the activation of this shareable image. If there is a mismatch, the

image activator aborts the activation and returns the error status SS$_SHRIDMISMAT unless the image is SYS$SSISHR, the system service intercept image, which is exempted.

10. If the image header indicates that the shareable image has an initialization section, the image activator sets the IMCB$V_INITIALIZE flag and records the address of the initialization section in IMCB$L_INITIALIZE.
11. The image activator performs the system version compatibility check described in step 14 of Section 8.3.1, using EXE$CHECK_VERSION and the system version override file.
12. The image activator creates a P1 copy of the KFERES for images installed resident.
13. The image activator processes the EISDs for each section in the shareable image.

    - If the EISD is a global EISD, representing a different shareable image, the image activator compares the portion of its name designating the image (that is, without the trailing *_nnn*) to the name of the IMCB most recently added to the work list. If the names are the same, the image activator does not add an IMCB to the work list. The comparison prevents some IMCB redundancy in the work list.

      If the names are different, the image activator creates an IMCB to describe the image. Before adding it to the work list, the image activator examines all existing work list entries for an entry whose name matches. If there is no match, the current IMCB is inserted at the head of the work list. Otherwise, the IMCB is inserted in place of the matching IMCB, and the matching IMCB is moved to the head of the work list.
    - If the EISD is not a global EISD, the image activator maps the section into process address space. Step 18 in Section 8.3.1 describes the processing of section EISDs and demand zero EISDs.

An image section flag that deserves special mention is EISD$V_VECTOR. If the EISD$V_VECTOR flag is set, the EISD identifies a private or global message section, a section containing a privileged library vector that describes inner mode system services and rundown routines in the image, or a system service intercept image.

- For a message section, the image activator validates the section, adds the message vector to the list maintained for the process, and sets IMAGCTX$V_SETVECTOR to indicate that vectors were added. The vectors are not usable until the end of image activation, when they are made accessible to the dispatcher.
- For a privileged library vector, the image activator checks that the image has been installed /PROTECTED and /SHARED and, if not, returns the error status SS$_PROTINSTALL. It records the address of the vector in IMCB$PS_PLV_ADDRESS.

- If the image section describes a system service intercept vector, the image activator checks that the image has been installed /PROTECTED and /SHARED and, if not, returns the error status SS$_PROTINSTALL. It records the address of the vector in IMCB$PS_SSI_PLV. This code is intended for the use of the system service intercept image, SYS$SSISHR.EXE, which is automatically activated by the image activator for images linked /DEBUG.

14. When all EISDs are disposed of, the image activator processes the quadword and longword relocation records in the image's fixup section, as described in Section 8.4.4. In earlier versions of OpenVMS, this step was performed by the $IMGFIX system service rather than by the image activator.
15. If a privileged shareable image was just activated, fixups are performed immediately for it.
16. If activating this image has added more IMCBs to the work list, this IMCB becomes the top of the stack maintained in the done list. The image activator goes to step 1 to process the next IMCB in its work list.

After the last IMCB has been processed, the image activator performs fixups for any partially protected images activated. It then performs the end processing described in Section 8.3.3.

### 8.3.3 Image Activator End Processing

If a main image was activated, the image activator performs the complete end processing described in this section. For a merged activation, it performs only steps 3, 4, 9, and 10.

The image activator's end processing consists of the following steps:

1. The image activator tests whether the image was linked with an image I/O segment larger than the standard space allocated during process creation. The standard size is determined by the SYSGEN parameter IMGIOCNT, which has a default value of 64 pagelets. However, the default can be overridden at link time with the following line in the linker options file:

```
IOSEGMENT = n
```

   If an image I/O segment larger than the default value is requested, the image activator requests the $CRETVA system service to create a replacement image I/O segment.

   If a P0-only image is being activated, the image activator creates the image I/O segment at the high-address end of P0 space.

   It calls RM$SET, in module RMSRESET, to initialize the image I/O segment.
2. The address space for the initial kernel thread's user stack is created with the Expand Region ($EXPREG) system service. The usual location of the

user stack is at the low-address end of P1 space, where the automatic stack expansion can add user stack space as needed. The location of the user stack in P0-only images is at the high-address end of the P0 image.

The default size of the user stack is 20 pagelets. The following line in the linker options file can override this value:

```
STACK = n
```

The image activator creates a user stack with two extra pagelets for system use during exception processing in case the user stack is corrupted.

3. If the argument HDRBUF was specified, the image activator copies information from the image header to the user-supplied buffer. The transfer address array is part of the image header; the image activator fixes up the transfer addresses in the array for the use of the caller, which will ultimately transfer control using these addresses. As part of the activation of the DECthreads run-time library, PTHREADS$RTL, the image activator inserts the address of the DECthreads initialization routine, PTHREAD_MAIN, into the transfer address array, causing it to be entered before any initialization routine requested by the main image or any shareable image. In particular, it executes before LIB$INITIALIZE. Section 8.5.1 provides more information about the transfer address array and its use in image startup.
4. If the argument RETADR was specified, it stores the beginning and end of the address range into which the image and any associated shareable images were mapped.
5. Running in kernel mode, the image activator stores the address of the high end of the user stack in the CTL$AQ_STACK array. Reserving space for system use during exception processing, the image activator loads an address two pagelets below the high end of the stack into the user mode stack pointer (USP) processor register. This is the value loaded into the stack pointer (SP) register when a CALL_PAL REI instruction returns the process to user mode, which usually occurs following the return from the image activator.
6. The privileges that will be in effect while this image is executing are calculated. The process header (PHD) field PHD$Q_IMAGPRIV is initialized to zero or the contents of KFE$Q_PROCPRIV, depending on whether an image installed with privileges is being activated. PHD$Q_IMAGPRIV is ORed with the process-permanent privilege mask at location CTL$GQ_PROCPRIV.

   The result is stored in the process privilege mask in the access rights block (ARB) at offset ARB$Q_PRIV (also known as PCB$Q_PRIV) and in PHD$Q_PRIVMSK. The uses of the various privilege masks are described in Section 8.8.1.

7. The image activator stores the address of the image header buffer in the global location CTL$GL_IMGHDRBF.
8. It checks whether image accounting was requested for this particular image or enabled for the system as a whole. If so, the image activator records various statistics, such as current CPU time, in their P1 locations.
9. If a known image is being activated, its use count (KFE$L_USECNT) is incremented. If the image was installed /OPEN, the share count in its WCB is incremented. If the image was installed /RESIDENT, the image activator also increments KFE$L_REF_COUNT. The image activator then sets the done bit in the IMCB to indicate that it has been activated. The actions in this step are done for each image being activated.
10. At this point, the image activator has finished its work. It releases its lock on the known file list, loads a final status into R0, and returns to its requestor. The requestor (EXE$PROCSTRT, LIB$FIND_IMAGE_SYMBOL, or a CLI) requests the $IMGFIX system service to perform address fixup, described in Section 8.4. When $IMGFIX returns, the requestor transfers control to the image as described in Section 8.5.

### 8.3.4 Computing the Proper Order of Image Initialization

As a by-product of its normal work, the image activator computes the order of initialization for multiple shareable images activated by a main image. The basic rule for image initialization is that if shareable image A calls shareable image B, then the initialization routine for image B must be called before the initialization routine for image A. This rule enables image A to call any routine in image B (or in any image that B calls) during A's own initialization.

The initialization routine for each activated image is called as part of image fixup (see Section 8.4.11). $IMGFIX first calls the initialization routine specified by the IMCB that is at the tail of the done list. It proceeds toward the head of the done list. The image activator must create the correct order of IMCBs in the done list by careful placement of IMCBs on both the work and done lists.

If code in image A calls code in image B, then at some point during the activation of image A, the image activator encounters a global EISD that references image B. The image activator builds an IMCB to insert at the head of the work list. Inserting these IMCBs at the head of the list ensures that these called, or son, images will be activated after the calling, or parent, image and generally before any other siblings of the parent.

Before actually inserting the IMCB into the work list, the image activator examines existing work list entries. If it finds an entry whose name matches that of the IMCB to be added, it inserts the IMCB after the matching IMCB and then moves the matching IMCB to the head of the work list. Since an

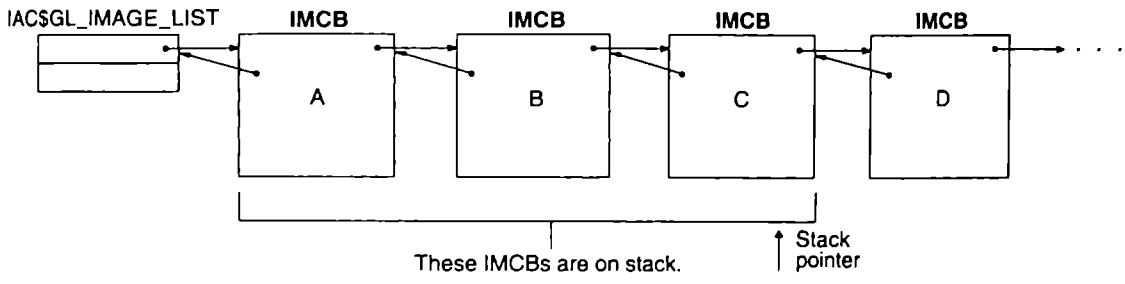



**Figure 8.17**
IMCB Stack in the Done List

image is only activated once no matter how many times it is referenced, this ensures that its mapping is controlled by the top-level accessor. Otherwise, the current IMCB is inserted at the head of the work list. This list generates a walk of the image call graph known as a preorder traversal.

A stack, implemented at the tail of the done list, is used to convert the preorder traversal for image activation into a postorder traversal for image initialization. Basically, a parent node remains on the stack until its last son is activated. A stack pointer points to the top of this stack in the done list. (Initially, the stack pointer points to the back link in the queue header.) Figure 8.17 shows how the IMCBs at the head of the done list form this stack.

To pop this stack, the stack pointer is simply moved to the left. The next IMCB from the work list is always inserted to the right of the top of the stack. It becomes the new top of the stack if it has any sons. IMCBs to the right of the top of the stack are always in the proper initialization order. IMCBs at and to the left of the stack pointer are parent IMCBs that still have descendants that have not been activated.

The stack is built to ensure that the sons and descendants of an image are always placed into the done list to the right of the IMCB of the parent. Since the done list is processed in reverse order during initialization, this placement ensures that all images called directly or indirectly by some image are initialized before that image itself.

The manipulation of the work and done lists is controlled by the IMCB$L_ACTIVE_SONS count in each IMCB. This field specifies how many of the image's sons have not yet been activated (their IMCBs are still in the work list) and how many have been activated but still have active sons of their own (these IMCBs are on the stack in the done list). The IMCBs to the right of the stack pointer in the done list have no active sons.

The following steps describe the image activator's manipulation of IMCBs in the work and done lists to generate the proper initialization order. The details of image activation are described in Sections 8.3.1 and 8.3.2 and are not repeated here. These steps are in routines IMG$DO_WORK_LIST and PROCESS_ISD_LIST, in module IMGMAPISD.

1. The image activator attempts to remove an IMCB from the front of the work list. If there is none, it goes on to end processing (see Section 8.3.3).
2. If this image has already been activated (the image is in the done list) and it still has active sons, then the image activator has detected a circularity. (The image is one of its own descendants, so no initialization order is possible.) In this rare case, all the images in the done list that are involved in the circularity must be marked. An error will be reported if a subsequent attempt is made to initialize one of those images. The images involved in the circularity are those IMCBs on the stack from the top of the stack down to and including the previously activated image.

   Regardless of whether there is a circularity, if the image was previously activated, the image activator deallocates the IMCB and continues at step 6.
3. Otherwise, this is a new image needing activation. The image activator inserts its IMCB just to the right of the top of the stack in the done list and zeros its IMCB$L_ACTIVE_SONS count.

   It then performs the detailed work of activation for this image (steps 3 through 16 in Section 8.3.2). During those steps, each time the image activator creates a new global IMCB (son), it places the new IMCB at the front of the work list and increments IMCB$L_ACTIVE_SONS in its parent's IMCB. After the parent image is activated but before its sons have been, this field contains the total number of shareable images referenced by the image.
4. If the field IMCB$L_ACTIVE_SONS in the IMCB to the right of the top of the stack is nonzero after the image has been activated, the image activator makes that IMCB the top of the stack and continues with step 1. This new parent remains on the stack until all its sons, which are located at the front of the work list, are activated and no longer have active sons of their own.
5. Otherwise, the field IMCB$L_ACTIVE_SONS in the IMCB to the right of the top of the stack is zero, and the image activator continues with step 6.
6. This step is called a "decrement parent" operation. IMCB$L_ACTIVE_SONS in the parent IMCB at the top of the stack must be decremented to indicate that one of its sons has been activated. If its count becomes zero, this same step must be repeated for its parent, and so on.

   If the stack is empty, there is no parent to decrement. The image activator continues with end processing (see Section 8.3.3). Otherwise, it decrements IMCB$L_ACTIVE_SONS in the IMCB at the top of the stack.
7. If the count is still positive (the image still has active sons), the IMCB remains at the top of the stack and the image activator continues with step 1. Otherwise, if IMCB$L_ACTIVE_SONS is now zero, the image ac-

tivator must decrement the IMCB$L_ACTIVE_SONS field in the parent of the IMCB.

8. When it reaches the IMCB at the top of the stack (the IMCB that initiated the activations and therefore has no parent), the image activator proceeds to its end processing. Otherwise, the image activator pops the stack by moving the stack pointer to the left in the done list and repeats step 6.

### 8.3.5 Example of Activation

The details of activating an image linked with several shareable images can be illustrated with an example. The example main image references the shareable images A and LIBRTL, image A references the shareable images B and LIBRTL, and image B references LIBRTL.

At the beginning of the activation, an IMCB representing the main image is placed into the work list. This first IMCB is moved from the work list to the done list. As its EISDs are processed, work list items are added for A and LIBRTL as the result of references in the main image.

| *Work List* | *Done List* | *Stack Top* |
|---|---|---|
| LIBRTL (main image) | Main image (2 sons) | ← |
| A (main image) | | |

After mapping the sections of the main image, the image activator removes the IMCB for LIBRTL from its work list. The EISD is processed and the main image's son count is decremented. Since LIBRTL has no sons, the main image remains at the top of the stack.

| *Work List* | *Done List* | *Stack Top* |
|---|---|---|
| A (main image) | Main image (1 son) | ← |
| | LIBRTL | |

The image activator removes the IMCB for image A from its work list. In processing A, work list items are added for B and LIBRTL. Since A has sons, it becomes the new stack top.

| *Work List* | *Done List* | *Stack Top* |
|---|---|---|
| LIBRTL (A) | Main image (1 son) | |
| B (A) | A (2 sons) | ← |
| | LIBRTL | |

The image activator removes the IMCB for LIBRTL from the work list, discovers the duplication, and discards the entry, decrementing A's son count.

| *Work List* | *Done List* | *Stack Top* |
|---|---|---|
| B (A) | Main image (1 son) | |
| | A (1 son) | ← |
| | LIBRTL | |

The image activator removes the IMCB for image B from its work list. In processing B, a work list item is added for LIBRTL. Since B has a son, it becomes the new stack top.

| *Work List* | *Done List* | *Stack Top* |
|---|---|---|
| LIBRTL (B) | Main image (1 son) | |
| | A (1 son) | |
| | B (1 son) | ← |
| | LIBRTL | |

The image activator removes the IMCB for LIBRTL from the work list, discovers the duplication, and discards the entry, decrementing B's son count. Since this brings B's count to zero, A (B's parent) becomes the stack top and its son count is decremented, again to zero. Thus the main image becomes the stack top, its count is decremented to zero, and the image activator performs its end processing. The done list is left in the correct order for image initialization.

| *Work List* | *Done List* | *Stack Top* |
|---|---|---|
| | Main image | ← |
| | A | |
| | B | |
| | LIBRTL | |

### 8.3.6 Other Cases of Image Activation

There are several other cases for which the image activator must check:

- Image activation at system initialization time. During initialization of the system, image files must be opened without the support of either RMS or the file system. The image activator calls special executive code that performs the simpler file system operations in the absence of a file system. These routines are briefly described with system initialization in Chapters *Bootstrap Processing* and *Operating System Initialization and Shutdown*.
- Merged image activation. A merged image activation occurs subsequent to the activation and transfer of control to a main image. This can be used for mapping a debugger, the Image Dump utility, the traceback handler, a message file, or a CLI into an unused area of P0 or P1 space. It is also used to activate a shareable image when an already activated image calls the Run-Time Library procedure LIB$FIND_IMAGE_SYMBOL.

  Rather than using the virtual address descriptors found in the merged image, the image activator simply uses the next available portion of P0 or P1 space. The user stack and image I/O segment are not mapped for a merged image. The RMS initialization routines are not called either,

because an image that is already executing has RMS context that cannot be destroyed.

- Message sections. These sections add per-process or image-specific entries to the message facility.
- P0-only images. The linker can produce images that map all temporary structures, including the user stack and the image I/O segment, in P0 space. The image activator must recognize this type of image and correctly map these two structures, usually located in the lowest address portion of P1 space.

  A P0-only image executes when the permanent part of the low-address end of P1 space must be extended without overwriting image structures. For example, the SET MESSAGE command causes a P0-only image called SETP0.EXE to execute. This image maps the indicated message section into the low-address end of P1 space and alters location CTL$GL_CTLBASVA to reflect the new boundary between the temporary and permanent parts of P1 space. This last step is critical if the message section is to remain mapped when later images terminate.
- Activation of a known image. When the image activator opens a known image, RMS places the address of the KFE in the CTX field of the FAB.

  The activation of a known image proceeds in the same way as that of a regular image, although some of the work that the image activator must perform in the regular case is avoided. In particular, a known image with a resident header is activated more quickly, because the header read operation is avoided.

  In any case, the EISDs must still be processed and the PTEs set up so that the image can execute. In addition, the image activator must update the usage statistics for this known image.
- Activation of a CLI. When the image activator determines that it is attempting to activate a CLI as a main image, it activates instead the image LOGINOUT. First, the image activator closes the CLI image file, because LOGINOUT performs its own file open. It clears IMCB$L_FLAGS and IMCB$L_IHD. Then it activates LOGINOUT and transfers control to it. LOGINOUT maps the CLI into P1 space and transfers control to it. Chapter 11 describes this flow.
- Images that do not reside on a random access mass storage device. The image activator can activate images from sequential devices (certain magnetic tape devices) and images located on another node of a network. An address space large enough to contain the entire image body is first created. The pages of the image body that are not demand zero are then copied into this address space, thus requiring all image body pages, including read-only pages, to be set up as writable. Debug symbol table information is ignored for such an image.

- Activation of an image resident in a granularity hint region. The EXE sections of such an image have already been activated in a system space granularity hint region. Global sections have been created for its shareable NOEXE sections. The image activator must map the nonshareable image sections in process-private space.

  When the image activator discovers that such an image is being activated, it sets IMCB$V_DISCONTIGUOUS and, if the image has been linked /SECTION_BINDING=(DATA,CODE), sets IMCB$V_COMPRESS_DATASEC as well to indicate that the process-private data sections should be compressed. It copies the KFERES to space allocated in the P1 allocation region. If data sections are to be compressed, it allocates a KFERES large enough to describe each of the process-private image sections that will be created.

  As the image activator processes the image's EISDs, if it encounters an EXE section, it confirms that the section has been mapped in system space already. When it encounters a shareable NOEXE section, it requests the $MGBLSC system service to map the global section. The image activator processes the other types of EISD as described in step 18 of Section 8.3.1. If data sections are being compressed, it bases each data section at the next available page and records information about the section in the corresponding KFERES substructure.
- Activation of an image installed with shareable address data. Shareable address data is currently supported only for resident shareable images, not for executable images. In addition to the actions described for resident images, the image activator sets IMCB$V_SHARE_LINK. It expects a KFERES section descriptor for each EISD and uses the address information from the KFERES to determine the section's base address in system or P1 space.

  When the Install utility processed the image, it created a system global section for its linkage section. All fixups to the linkage section were processed at that time. Since there is no longer a need for write access to the linkage section, the image activator ignores the WRT attribute in the image's linkage section EISD and maps it read-only.

  The Install utility also created a subset, or compressed, fixup section in system space describing all fixups remaining after those for the linkage section were processed, for example, protection-change fixups and fixups for process-private sections. The image activator creates a P1 space copy of the compressed fixup section and stores its address in the KFERES and the IMCB. It is later used by the $IMGFIX system service.
- Privileged shareable images. User-written system services and rundown routines are implemented as privileged shareable images. System service procedures that are not part of an executive image (for example, $MOUNT and $DISMOU) are implemented as privileged shareable images. A privileged shareable image contains a privileged library vector (PLV). The PLV

for a privileged shareable image with user-written system services or rundown routines lists the kernel and executive mode system services and routines in the privileged shareable image.

After processing all the EISDs in a privileged shareable image or a partially protected image, the image activator calls the routine IMG$PRVSHRIMG, in module SYSIMGFIX, to generate procedure descriptors and transfer routines for the services. If the image is completely protected, the routine checks that no shareable images other than SYS$PUBLIC_VECTORS and SYS$BASE_IMAGE have been linked with it; if any has, it returns the error status SS$_NOSHRIMG.

IMG$PRVSHRIMG creates enough virtual address space to contain system service transfer routines and their procedure descriptors. It processes the PLV as described in Chapter *System Service Dispatching* and then does fixups and protection changes as described in Section 8.4. If a system service intercept image is being activated, it contains a private copy of the SYS$PUBLIC_VECTORS symbol vector plus its own additional vectors. The IMCB for SYS$PUBLIC_VECTORS is located or created, and its IMCB$PS_SYMBOL_VECTOR_ADDRESS is pointed to the private symbol vector copy. If the image has an initialization procedure, IMG$PRVSHRIMG calls it.

- Activation of interoperable images. The OpenVMS Alpha system supports the execution of translated OpenVMS VAX images through a run-time environment called the Translated Image Environment (TIE). Moreover, it allows a combination of native and translated images to be activated together. Procedure calls between the two types of image require additional support because the OpenVMS Alpha and OpenVMS VAX calling standards differ. An automatic jacketing mechanism detects cross-architecture calls and makes the conversions between the two calling standards. This mechanism is implemented partly by the image activator.

  After processing the entire work list, the image activator checks whether any translated images were activated. If so, it must create virtual address space for the structures used in transforming calls from native images to procedures in translated images. It scans the fixup section list for images whose fixup sections include any records for linkage pairs with procedure signature blocks. These represent potential calls from a native image to a translated shareable image. For such fixups, it calculates the amount of virtual address space required to create a bound procedure descriptor and, optionally, a nondefault procedure signature block. It requests the $CRETVA system service to create that address space and records its starting address in CTL$GL_BPD_PTR. The structures are built during image fixup processing, as described in Section 8.4.8.

  Chapter 9 describes the jacketing mechanism and the environment for translated images.

## 8.4 ADDRESS RELOCATIONS AND FIXUPS

An image can refer to locations within itself or within another image. Such references require adjustment based on the address space assigned to the images. The linker creates an image section called a fixup section in each image. The fixup section describes the locations in the image that contain addresses that must be adjusted after address assignment. The adjustment to an intra-image address reference is called a relocation. The location to be adjusted can be a quadword or a longword. The adjustment to a reference from one image to another, an inter-image reference, is called a fixup.

The $IMGFIX system service procedure, EXE$IMGFIX, in module SYSIMGFIX, performs address fixups after the image activator maps all necessary images. In earlier versions of OpenVMS, EXE$IMGFIX performed all relocations and fixups. Relocations now occur during image activation, and both relocations and fixups can occur during installation for images installed with shareable address data. Section 8.4.10 describes this optimization, designed to minimize relocation and fixup overhead for certain shareable images.

The module RELOCATE_AND_FIXUP contains the routines that perform relocation and fixup. In the following sections, "image fixup routines" refers to the routines in the RELOCATE_AND_FIXUP module, whether called by the Install routine INS$IMGFIX, by $IMGACT, or by EXE$IMGFIX.

### 8.4.1 Relocation and Fixup Processing

The different types of address reference that can require adjustment are as follows:

- Quadword and longword address references. The MACRO .ADDRESS directive (or its high-level language equivalent) references a fixed address in virtual memory. Resolution of a .ADDRESS reference to a location in a shareable image is deferred so that the shareable image need not be loaded at a fixed base address. .ADDRESS references are fixed up when the image is activated, after the starting addresses of its image sections have been determined.

  The MACRO .ASCID directive (or its high-level language equivalent) builds an ASCII string and a descriptor for it. It incorporates the equivalent of a .ADDRESS directive referencing the string. .ASCID directives within a shareable image are relocated after the base address of the shareable image has been determined. In the following sections, text references to .ADDRESS directives include those generated by .ASCID directives.
- Code address references. A code address directive references the entry point of a procedure rather than its procedure descriptor. Most code address directives are contained within procedure descriptors.
- Linkage pairs. A language compiler represents a linkage to an external procedure as a two-quadword data structure called a linkage pair. By the

time a call using a linkage pair is executed, the first quadword of the linkage pair must contain the external procedure's code entry address, and the second quadword must contain the address of its procedure descriptor. When the linker links an object module containing a call with the object module containing the called procedure, the linker fills in the linkage pair in the calling object module.

- Linkage pairs with procedure signature blocks. A compiler can generate a data structure called a procedure signature block to describe a call from a native module. The procedure signature block details the arguments passed to the target procedure. For an image linked with the qualifier /NONATIVE_ONLY, the linker builds the procedure signature blocks in the fixup section. Each is processed during image fixup if its target procedure is in a translated image.

A typical executable image requires no relocations because the linker assigns its address space, and it can calculate correct values for intra-image references. However, an executable image linked /SECTION_BINDING does contain relocation records. The relocation records enable its image sections to reside at base addresses other than those calculated by the linker. For instance, if installed /RESIDENT, the base address of its code image section is in system space rather than P0 space.

If an executable image references any shareable images, it requires fixups. Its fixup section describes the shareable images with which it was linked and lists the locations that must be adjusted after the shareable image is assigned address space.

Because the address space for a shareable image is not assigned until the image is activated, all intra-image address references within a shareable image must be relocated. If the shareable image is linked with other shareable images, its references to addresses in them must be fixed up. Its fixup section describes those shareable images and lists the locations requiring relocation or fixup.

When the $IMGACT system service encounters an EISD describing a fixup section, it stores the base address of the image being processed in the fixup section at EIAF$L_BASE_VA and inserts the section at the head of the list pointed to by CTL$GL_FIXUPLNK. In the case of an image installed with shareable address data, this fixup section is a subset of the original fixup section, describing any relocations or fixups that could not be processed when the image was installed.

As each image is activated, the image activator performs the longword and quadword relocations. Relocations are performed first, before any fixups, because the relocations adjust addresses within symbol vectors that are used in fixups. After the main image and its tree of shareable images have been activated, the $IMGFIX system service performs fixups, using the list of fixup sections.

For an image installed with shareable address data, most of the relocation and fixup processing took place earlier, when the image was installed. A main image linked with such an image is activated faster, with fewer page faults. Section 8.4.10 describes this optimization.

Image sections that contain relocated or fixed up addresses must be writable during image activation so that the modifications can be made. If such a section is made up of NOWRT PSECTs, the protection on its pages must be altered after the modifications are made. The linker describes these and other protection changes in the fixup section as well.

The sections that follow describe the fixup section and its use by the image fixup routines.

### 8.4.2 Fixup Section

A fixup section describes all relocations and fixups to be made to the image and a list of the shareable images referenced by the image. Figure 8.18 shows the organization of an image and the layout of the fixup section within it.

The fixup section is made up of a fixed area and a number of variable-length areas. The following fields in the fixed area contain the byte offset relative to the start of the fixup section of each variable-length area:

- EIAF$L_QRELFIXOFF—Offset to the quadword relocation records

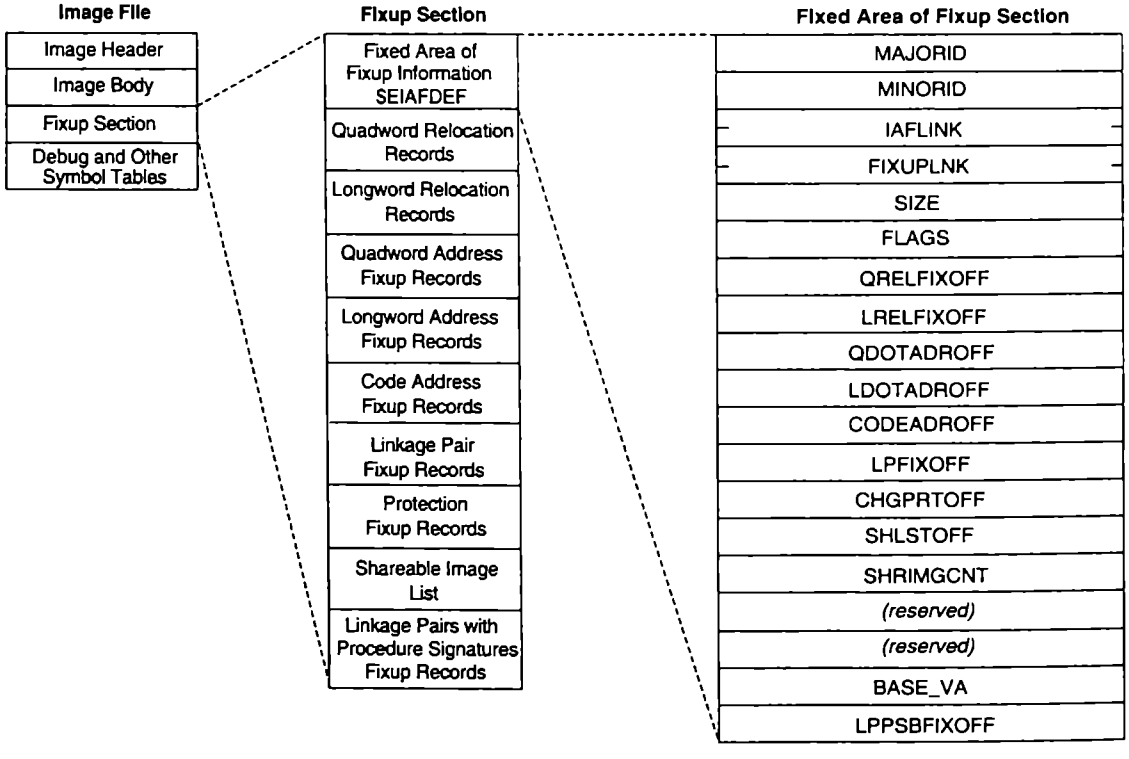



**Figure 8.18**
Image with Fixup Section Layout

- EIAF$L_LRELFIXOFF—Offset to the longword relocation records
- EIAF$L_QDOTADROFF—Offset to the quadword .ADDRESS fixup records
- EIAF$L_LDOTADROFF—Offset to the longword .ADDRESS fixup records
- EIAF$L_CODEADROFF—Offset to the CODE_ADDRESS fixup records
- EIAF$L_LPFIXOFF—Offset to the LINKAGE_PAIR fixup records
- EIAF$L_CHGPRTOFF—Offset to the protection change fixup records
- EIAF$L_SHLSTOFF—Offset to the shareable image list entries
- EIAF$L_LPPSBFIXOFF—Offset to the fixup records for linkage pairs with signature blocks

An offset value of zero means the corresponding relocations or fixups are not present in the image and thus the corresponding variable-length area is not present in the fixup section.

EIAF$L_MAJORID and EIAF$L_MINORID describe the version of the fixup section.

EIAF$L_FLAGS contains one defined flag, which, when set, identifies the image as a shareable image.

EIAF$L_SHRIMGCNT contains the number of shareable images with which the image has been linked, actually the number of entries in the shareable image list. In an OpenVMS VAX image, this count includes the image itself; in an OpenVMS Alpha image, the count typically does not. If, however, an image has defined universal symbols with alias names, the count includes the image itself.

The fields that follow have meaning only for an in-memory copy of a fixup section in an image being activated.

The two longwords at EIAF$Q_IAFLINK link the fixup section into a list of fixup sections from all activated images.

EIAF$Q_FIXUPLNK links the fixup section into a list of fixup sections yet to be processed. The listhead is at CTL$GL_FIXUPLNK.

EIAF$L_BASE_VA contains the address of the IMCB describing the image containing this section until the relocations and fixups have been done for the image. Then it contains the base address at which the image was activated.

### 8.4.3 Shareable Image List

EIAF$L_SHLSTOFF contains the offset to the shareable image list entries (SHLs) in the fixup section, and EIAF$L_SHRIMGCNT contains the number of SHLs. Each SHL identifies a shareable image referenced by the image. Moreover, each shareable image has its own fixup section and SHL list.

An SHL can also represent one of the system base images: an image that requests system services has an SHL representing SYS$PUBLIC_VECTORS.EXE; an image linked /SYSEXE to resolve executive universal symbols has an SHL representing SYS$BASE_IMAGE.EXE.

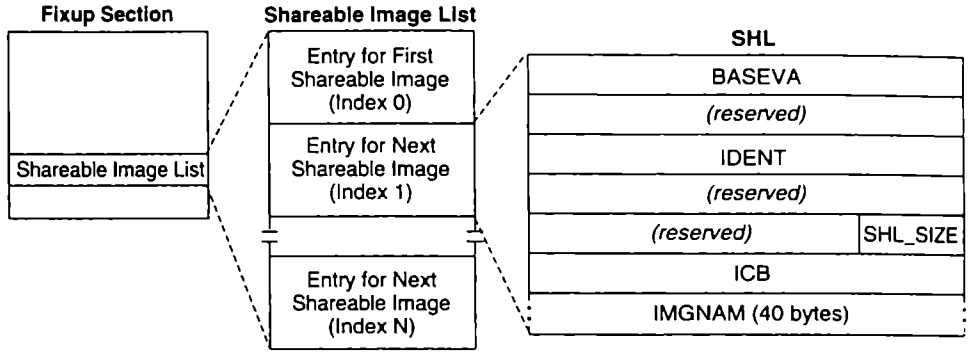



**Figure 8.19**
Shareable Image List

Although the linker builds the shareable image list, it initializes only the fields SHL$T_IMGNAM, SHL$L_IDENT, and SHL$B_SHL_SIZE.

Figure 8.19 shows the layout of an SHL and SHLs within a fixup section.

Each SHL contains the name of the shareable image it represents at SHL$T_IMGNAM. The image fixup routines use this name to match the SHL with an IMCB in the done list. They copy the virtual address of the shareable image's symbol vector from the IMCB into SHL$L_BASEVA. If the SHL represents one of the system base images, the image fixup routines copy the address of its symbol vector from the appropriate system cell instead.

SHL$B_SHL_SIZE contains the size of the SHL, 64 bytes.

SHL$L_IDENT contains the GSMATCH information from the version of the shareable image with which the image was linked. Its contents are copied from the match control information in the global EISD identifying the shareable image.

SHL$L_ICB contains the address of the IMCB that this SHL represents.

### 8.4.4 Relocation Records

An example of a location in a shareable image that needs relocation is a procedure descriptor, which contains the address of the code entry point for that procedure. After linking, such a location contains an image-relative offset. After the image is assigned address space, the base address of the image must be added to the offset. The fixup section contains relocation records that identify the locations in the image requiring that type of adjustment.

The linkage section and symbol vector contain many of the locations in an image that require relocation. Note that an image section containing relocatable addresses cannot be shared among processes, since the resolutions of those addresses are specific to the virtual address space in each process. Consequently, linkage sections and symbol vectors are not shareable unless the image is installed /SHARE=ADDRESS_DATA.

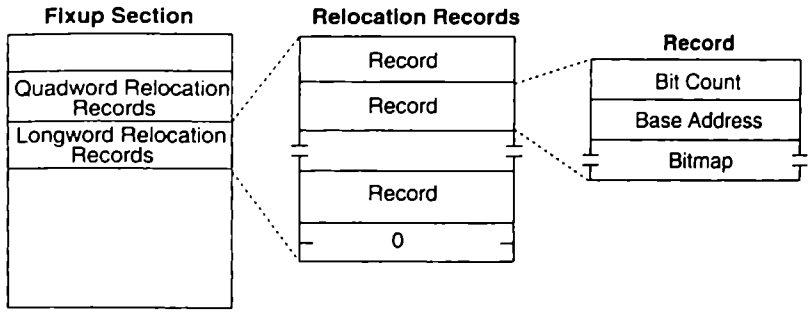



**Figure 8.20**
Relocation Records

Each image section that contains locations with relocatable addresses is represented by one or more relocation records. A relocation record specifies the base address in the image of a range of addresses, some of whose contents need relocation. Each relocation record contains a bitmap with one bit for each location in the address range. When set, a bit indicates that the corresponding location's contents need relocation. The bitmap length is rounded up to an integral number of quadwords. The linker generates enough relocation records so that the bitmaps do not contain large numbers of adjacent zeros; if a particular record does contain too many adjacent zeros, the linker breaks it into multiple records.

Figure 8.20 shows the layout of relocation records within a fixup section. In a quadword relocation record each bit represents a quadword, and in a longword relocation record each bit represents a longword. Each of the two sets of relocation records ends with a quadword of zero.

To perform the relocations in one record, the image fixup routines add the base address of the image to the image offset and scan the bitmap for set bits. For each set bit, they calculate the corresponding address whose contents require relocation and add the base address of the image.

For an image installed /RESIDENT, the image fixup routines first determine whether a range of addresses requiring relocations is within a resident section or a compressed data section and calculate the activated address requiring relocation based on the contents of the KFERES. If an address is within a resident or compressed data section, the new contents will be the old contents minus the image offset of that image section plus the address of the resident section.

### 8.4.5 Longword and Quadword Address Fixups

A longword address record or quadword address record (or both) exists for each shareable image that is the target of .ADDRESS or .ASCID directives. A longword address record records the locations of longwords in the image being fixed up, and a quadword address record, the locations of quadwords.

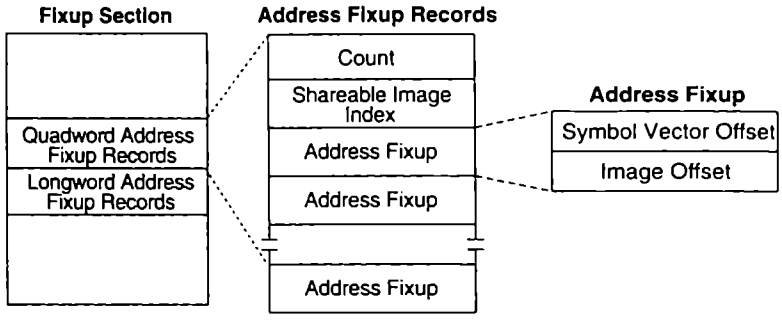



**Figure 8.21**
Longword and Quadword Address Fixups

Each record consists of an entry count; the index of the SHL associated with the shareable image, which contains the referenced data; and a series of fixups. Figure 8.21 shows the layout of longword and quadword address fixup information within a fixup section.

Each fixup is a pair of longwords. The first contains the symbol vector offset of the target symbol, and the second, the offset into the image being fixed up of the longword or quadword. At the specified offset in the image is a longword or quadword containing zero or a constant to be added to the fixed-up address.

For each record, the image fixup routines typically perform fixups as follows:

1. They use the index into the shareable image list to locate the SHL associated with the shareable image.
2. From the SHL, they obtain the address of the shareable image's symbol vector.
3. For each address fixup, they add the image offset to the base address of the image being fixed up to get the address of the longword or quadword to be fixed up. To the contents of that longword or quadword, they add the contents of the address portion of the symbol vector linkage pair at the specified offset in the shareable image symbol vector.

   For an image installed /RESIDENT, the image fixup routines first determine whether an address requiring a fixup is within a resident section or a compressed data section and, if so, calculate the activated address requiring fixup based on the contents of the KFERES.

### 8.4.6 Code Address Fixups

There is a code address fixup record for each shareable image that is the target of code address directives. Each record consists of an entry count, the index of the SHL associated with the shareable image, and a series of offsets into the image being fixed up. Figure 8.22 shows the layout of code address fixup information within a fixup section.

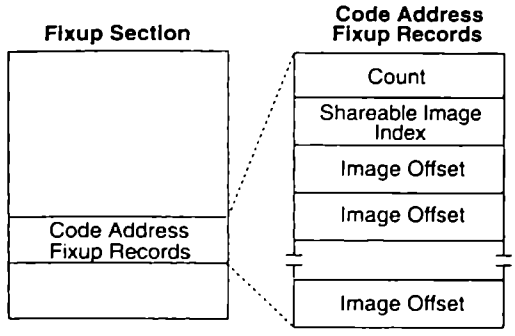



**Figure 8.22**
Code Address Fixups

At the specified offset in the image is a quadword containing the symbol vector offset of the symbol referenced. To perform each fixup, the image fixup routines calculate the address of the location in the image being fixed up and examine the quadword at that address to get the symbol vector offset of the target address. They store the code address portion of that symbol vector linkage pair in the specified quadword.

For an image installed /RESIDENT, the image fixup routines first determine whether an address requiring a fixup is within a resident section or a compressed data section and, if so, calculate the activated address requiring fixup based on the contents of the KFERES.

### 8.4.7 Linkage Pair Fixups

A linkage pair fixup record exists for each shareable image that is the target of linkage pairs in the image being fixed up. Each record consists of an entry count, the index of the SHL associated with the shareable image, and a series of offsets into the image being fixed up. Figure 8.23 shows the layout of the linkage pair fixup area within a fixup section.

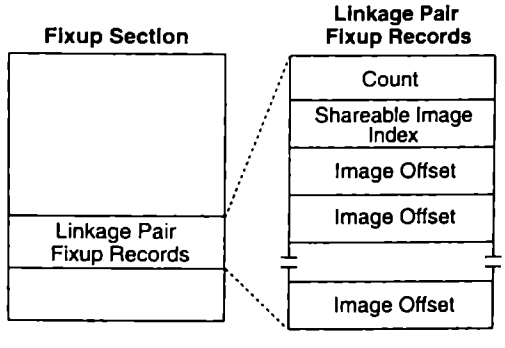



**Figure 8.23**
Linkage Pair Fixups

At the specified offset in the image is a linkage pair, the first quadword of which contains the symbol vector offset of the target symbol. To perform each fixup, the image fixup routines calculate the address of the location in the image being fixed up and examine the quadword at that address to get the symbol vector offset of the target address. They store the linkage pair at that symbol vector entry in the specified linkage pair.

For an image installed /RESIDENT, the image fixup routines first determine whether an address requiring a fixup is within a resident section or a compressed data section and, if so, calculate the activated address requiring fixup based on the contents of the KFERES.

### 8.4.8 Fixups for Linkage Pairs with Procedure Signature Blocks

When the modules of a native image are compiled /TIE and the image is linked /NONATIVE_ONLY, code in that image can call procedures in both native and translated images. A call to a translated image has an associated procedure signature block and goes through a jacket procedure. The jacket procedure uses the signature block (see Chapter 9) to transform the native arguments into arguments appropriate for the translated image.

There is a fixup record for linkage pairs with procedure signature blocks for each shareable image that is the target of linkage pairs in the image being fixed up. Each record consists of an entry count, the index of the SHL associated with the shareable image, a series of entries, and a series of procedure signature blocks. Each entry contains an offset into the image being fixed up and the unrelocated address of the corresponding procedure signature block if there is one. If there is not, that is, if the procedure signature is the default one, the entry contains zero instead.

Figure 8.24 shows the layout of the fixup records for linkage pairs with procedure signature blocks.

At the specified offset in the image is a linkage pair, the first quadword of which contains the symbol vector offset of the target symbol. In processing each record, the image fixup routines first determine whether the shareable image is native or translated. If native, they essentially perform a simple linkage pair fixup as described in the previous section.

If the shareable image is translated, for each fixup the image fixup routines calculate the address of the location in the image being fixed up and save the contents of the quadword at that address, the symbol vector offset of the target procedure in the translated image.

In the virtual address space the image activator created for bound procedure descriptors and procedure signature addresses (see Section 8.3.6), the image fixup routines build a bound procedure descriptor to represent a transfer to EXE$NATIVE_TO_TRANSLATED_BP, in module SYSJACKET. The target procedure value is the code entry point of the target translated procedure.

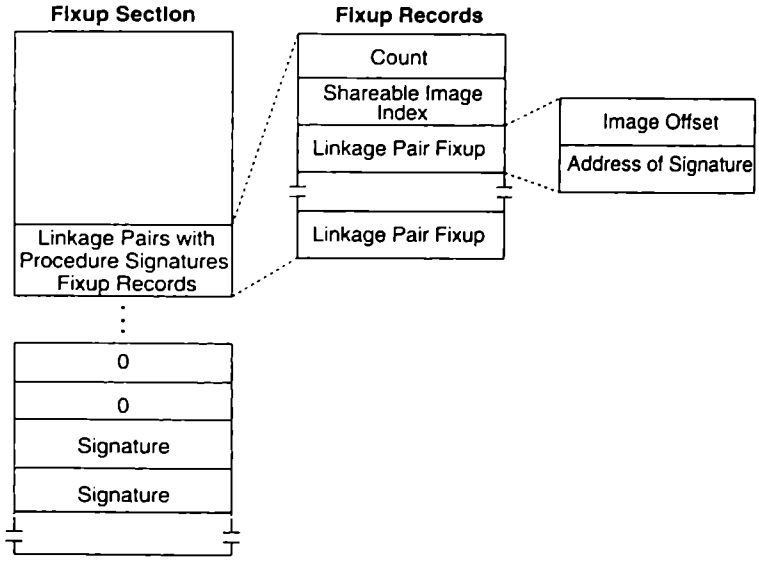



**Figure 8.24**
Fixups for Linkage Pairs with Procedure Signature Blocks

If there is a nondefault procedure signature block, the image fixup routines copy it immediately after the bound procedure descriptor and store its offset in the bound procedure descriptor.

They store the bound procedure descriptor address and the code entry address of the procedure EXE$NATIVE_TO_TRANSLATED_BP in the linkage pair of the image being fixed up. Later, when the image calls this procedure, control is transferred to the $NATIVE_TO_TRANSLATED system service, which uses the procedure signature block to build arguments for the translated image's procedure and then transfer to the translated procedure.

### 8.4.9 Page Protection Fixup

After address fixup is complete, the image fixup routines adjust page protection as specified in the page protection data area of the fixup section. For the image fixup routines to relocate or fix up a location requires that the page containing the location be writable. A read-only image section containing such a location is originally mapped as writable. The linker creates an entry in the page protection data area for each section of this type, specifying a new page protection of UR. After address fixup, the image fixup routines request the Set Protection on Pages ($SETPRT) system service for each entry in the page protection data area.

An image that contains a protected section, for example, a privileged shareable image containing user-written system services, has a page protection entry that alters the protection of that section to UREW.

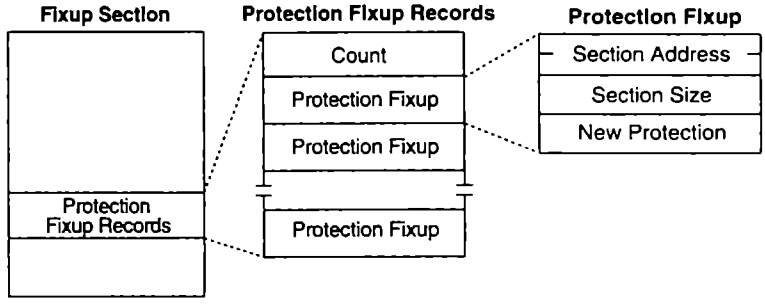



**Figure 8.25**
Page Protection Area

In an image linked with any shareable images, a final page protection entry alters the protection of the fixup section itself to UREW. The fixup section pages are protected from user mode modification so that the SHL entries remain unmodified. Figure 8.25 shows the layout of the page protection area.

### 8.4.10 Relocation and Fixup Optimization

In an executable image, the linker assigns a base address to each private image section. The Install utility can override these assignments for certain image sections, for instance, a code section in an image installed /RESIDENT. The image activator uses the addresses determined by the linker and Install utility to place the image section in memory.

The linker does not assign address space for shareable images. The linker, the $IMGACT system service, and the $IMGFIX system service cooperate to postpone shareable image address assignment from link time to image activation. Delaying address assignment increases the lifetime of an image by partly decoupling it from the shareable images with which it is linked. For example, a newer version of a shareable image can grow, shrink, or even refer to additional shareable images without requiring the relinking of an image that linked with it.

The cost of this flexibility is additional time and effort resolving address references at image activation time. Normally, relocation and fixup processing for an installed shareable image occurs each time a process activates a main image linked with the shareable image, since the address space range into which the shareable image will be mapped is not predetermined. Relocation and fixup processing requires the image's fixup section to first be paged into memory. Each fixup is then processed by determining the image address that must be written with new address information, faulting that page into memory, and writing the updated information. The costs include paging activity, CPU time, and physical memory.

In an effort to reduce this overhead for frequently activated shareable images, the Install utility provides a way to preactivate selected images into a

fixed address range, so that relocations and fixups need only be performed once. Installing an image /RESIDENT/SHARE=ADDRESS_DATA causes the Install utility to assign any image section not mapped in a granularity hint region to a permanent location in P1 space. All processes activating the shareable image will map its nonresident sections in the same P1 space addresses. Therefore, address relocation for the shareable image need only occur at installation. Most address fixups can be preprocessed as well.

The linkage section is the image section to which the majority of relocations and fixups apply. Thus, it is normally a process-private image section, written when fixup processing occurs and set read-only by a protection change fixup at the end of image activation. However, when all processes map all of an image into the same resident locations, the linkage section can be shared, reducing the demand for physical memory. The Install utility creates a system global section for it and processes relocations and fixups. Subsequent image activations merely map the global section read-only. Thus, this Install preactivation feature is sometimes referred to as shareable linkage.

The P1 space image region into which nonresident image sections are mapped is located by CTL$A_IMGREG and managed by the Install utility. Its size is controlled by the SYSGEN parameter IMGREG_PAGES. The Install utility also manages a system IMCB list whose listhead is located at IMG$GL_IMAGE_LIST. Each IMCB inserted on this list represents a resident image installed with shareable address data.

Restrictions are placed on the manner in which such images are linked and the order in which such images are installed. Shareable address data is currently supported only for resident shareable images, not for executable images, and the image must be linked with the /SECTION_BINDING qualifier. It can only link against shareable images that will be installed with shareable address data before the installation of the image linking against them.

By default both OpenVMS and DECwindows install commonly used shareable images in the region during system startup, so that all processes can take advantage of the improved performance. The OpenVMS shareable images currently installed with shareable address data are DECC$SHR, DPML$SHR, CMA$TIS_SHR, LIBRTL, and LIBOTS.

### 8.4.11 Operation of EXE$IMGFIX

The $IMGFIX system service procedure, EXE$IMGFIX, in module SYSIMGFIX, performs address fixups after the image activator maps all necessary images. It calls routines from the module RELOCATE_AND_FIXUP to perform the actual fixups.

EXE$IMGFIX processes entries from the fixup section list created by EXE$IMGACT. It first scans the SHL list in each fixup section and confirms

that each shareable image referenced by a fixup section has been activated, returning the error status IMG$_IMAGE_NOT_FOUND if not. It records the address of the image's symbol vector in SHL$L_BASEVA and its IMCB address in SHL$L_ICB. It then processes the fixup section list, first performing all fixups and then all protection changes.

In the case of an image installed /RESIDENT with compressed data sections, EXE$IMGFIX and the procedures it calls must transform image offsets into their equivalent virtual addresses using the KFERES structure.

Following address fixup and page protection modification, EXE$IMGFIX tests whether any privileged shareable images have been activated. If so, it requests the $IMGACT system service, specifying the IAC$V_SETVECTOR flag. Running in executive mode, the image activator initializes the P1 space dispatch vectors for user-written system services, rundown routines, and message sections.

If any shareable image specified an initialization routine, EXE$IMGFIX scans the done list, IMCBs representing activated images, from back to front. Running in user mode, EXE$IMGFIX calls the initialization routine of each shareable image that specified one.

## 8.5 IMAGE STARTUP

EXE$PROCSTRT or a CLI can request image activation and fixup, as described in Chapter 11. After successful image activation and fixup, the image is called at its transfer address. Depending on how the image was linked, the initial transfer of control may be to a debugger, a user-supplied initialization procedure, or the executable image itself.

### 8.5.1 Transfer Address Array

In addition to the EISDs previously discussed, the linker includes in the image header a data structure called a transfer address array. This array contains the user-supplied transfer address. It also provides the means for including a debugger or traceback handler in the user image and for specifying that the image contains initialization code.

The layout of the transfer address array is pictured in Figure 8.26, along with several examples of its contents.

EIHA$L_SIZE contains the size of the array, always 48 bytes. For a main image the array can potentially contain three transfer addresses:

- Typically, the first transfer address represents the $IMGSTA system service, sometimes called the debugger bootstrap. Unless the image has been linked /NOTRACEBACK or /DEBUG=*file-spec*, the low-order longword of the first transfer address contains the offset of SYS$IMGSTA in the SYS$PUBLIC_VECTORS symbol vector, fixed at $340_{16}$. EXE$IMGACT in-

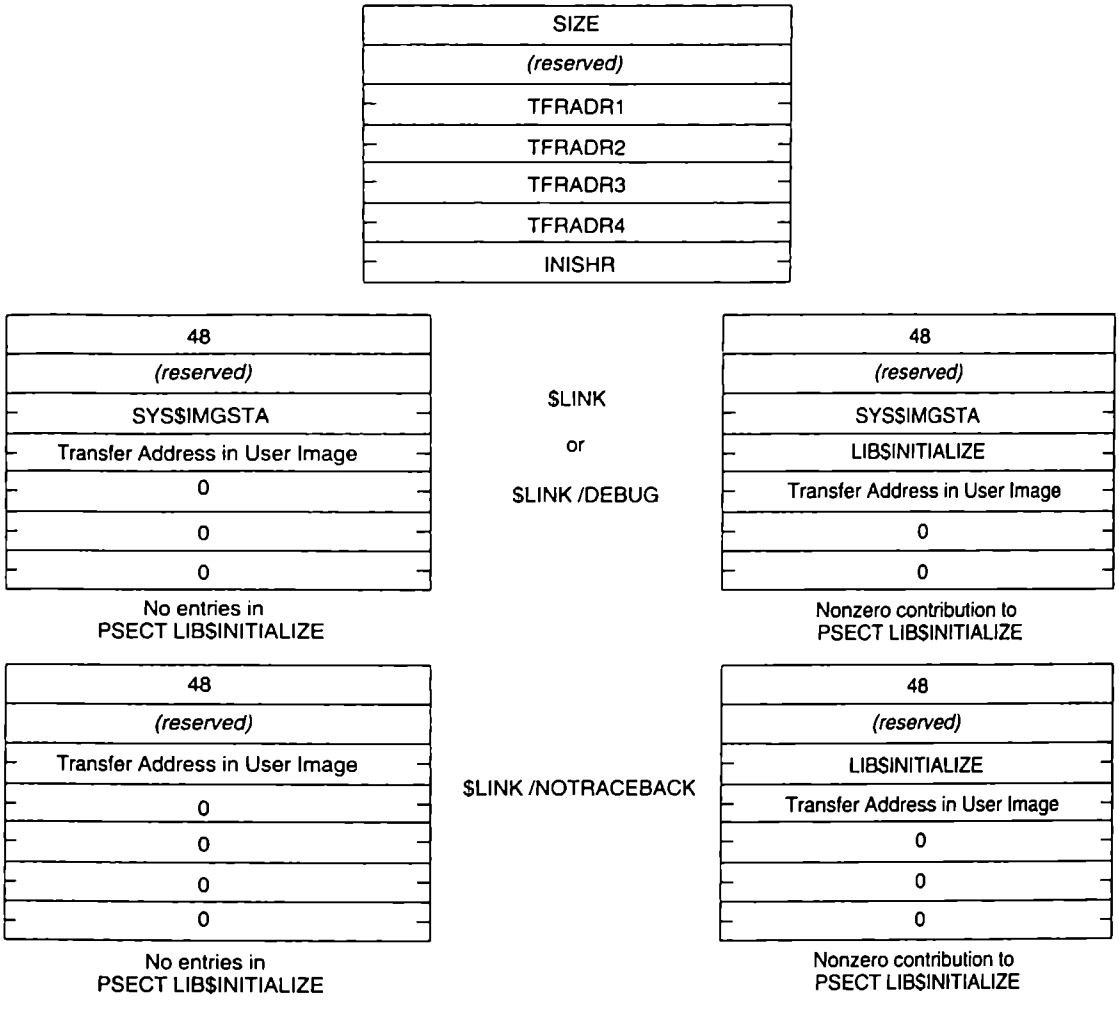



**Figure 8.26**
Layout of a Main Image Transfer Address Array

terprets the value FFFFFFFF $00000340_{16}$ to mean SYS$IMGSTA and performs the necessary fixup.

- The address LIB$INITIALIZE occurs next for a main image that contains initialization procedures. This address in the image file is image-relative and is relocated by EXE$IMGACT when the image is activated.

  If an image uses DECthreads, the image activator shifts the LIB$INITIALIZE entry and any subsequent entries down one location. It inserts the address of the initialization routine PTHREAD_MAIN, in the DECthreads run-time library PTHREADS$RTL, into the transfer address array in place of LIB$INITIALIZE. This allows thread initialization to occur before any initialization routine requested by the main image.
- The next transfer address is the one associated with the user image, either the argument of a .END directive for a MACRO-32 or MACRO-64

program or the first statement of a main program written in a high-level language. This address in the image file is image-relative and is relocated by EXE$IMGACT when the image is activated.

- A final entry containing zero is the end-of-list indication, no matter what options were passed to the linker.

Each initialization procedure called through the transfer address array is responsible for calling the next procedure in the array.

Each module that is part of a main image and that contains an initialization procedure contributes a table entry with the procedure value to a PSECT called LIB$INITIALIZE. One or more modules in the main image must declare as external the symbol LIB$INITIALIZE. The presence of that declaration results in the linker's adding to the image a STARLET.OLB object module that contains the procedure LIB$INITIALIZE. The initialization transfer procedure LIB$INITIALIZE and the LIB$INITIALIZE PSECT are described further in the *OpenVMS Programming Concepts Manual*.

If the DCL command LINK/DEBUG=*file-spec* is used to link an image, the explicit file specification is the name of a particular debugger object module to be linked with the image. The linker places the transfer address found in the specified debugger file into the first element in the transfer address array rather than the quadword representing SYS$IMGSTA. It puts the image transfer address into the second element of the array. The debugger object module must perform SYS$IMGSTA's role in transferring control to the image.

If the /NOTRACEBACK qualifier is included (and not overridden implicitly by including an explicit /DEBUG qualifier), then there is no debug transfer address. In all other cases (including the DCL command LINK/DEBUG, which does not specify an explicit debugger module), the linker places the address of SYS$IMGSTA in the first element of the transfer address array.

For a shareable image, the last element in the array, at offset EIHA$Q_INISHR, contains the procedure value of an initialization routine if one exists.

### 8.5.2 $IMGSTA System Service

Except for an image linked with the /NOTRACEBACK or the /DEBUG=*file-spec* qualifier, every main native image executes the $IMGSTA system service. The system service procedure, EXE$IMGSTA in module SYSIMGSTA, runs in user mode. This procedure examines link and CLI flags to determine whether to start the image directly or to map the debugger into P0 space and transfer control to it.

EXE$IMGSTA first tests whether it should map a debugger into P0 space. The mapping is done if either of the following conditions is true:

- If the program was linked with the DCL command LINK/DEBUG and simply run (that is, not run with a RUN/NODEBUG command)
- If the program was run with the DCL command RUN/DEBUG, independent of whether the debugger was requested at link time

The debugger is not mapped if the image was run with a RUN/NODEBUG command or if the /DEBUG qualifier was omitted from both the LINK command and the RUN command.

If a debugger is to be mapped, EXE$IMGSTA requests the Translate Logical Name ($TRNLOG) system service to translate the logical name LIB$DEBUG. If there is no translation, EXE$IMGSTA uses the string DEBUG as the debugger name. EXE$IMGSTA then requests the $IMGACT system service to activate the debugger image from the default device and directory SYS$LIBRARY. It specifies flags for a merged activation in P0 space, so that the debugger will be mapped at addresses just higher than the main image and its shareable images. EXE$IMGSTA then requests the $IMGFIX system service and finally transfers control to the debugger image through the third quadword from the beginning of the debugger image, which contains the procedure value of the debug entry point. The debugger initializes itself, establishes a breakpoint at the beginning of the image, and returns to EXE$IMGSTA.

If no debugger is mapped, EXE$IMGSTA establishes a condition handler for the current call frame. This condition handler, BOOT_HANDLER, gains control on signals that the image does not handle directly. After gaining control, the condition handler calls the debugger, calls the traceback handler, or resignals.

Whether or not a debugger is mapped, EXE$IMGSTA alters the arguments with which it was called to point to the next address in the transfer address array and passes control to the next transfer address. This is either the DECthreads initialization procedure, the Run-Time Library procedure LIB$INITIALIZE, or the transfer address of the user image.

### 8.5.3 Exception Handler for Traceback

BOOT_HANDLER, the condition handler established by EXE$IMGSTA before the image was called, has two main functions:

- It calls a debugger if a DEBUG command is typed after an image is interrupted with a CTRL/Y.
- If an unfielded condition occurs, it causes an image dump, if one was requested, and calls the traceback handler to produce a symbolic stack dump.

However the handler is entered, it first calls EXE$CATCHALL_EXTENSION, in module PROCSTRT, which calls any catch-all condition handler extensions declared by the image (see Chapter *Condition Handling*).

If a user interrupts execution of a nonprivileged image by typing CTRL/Y and DEBUG, the DCL CLI generates the signal SS$_DEBUG. (Privileged images are simply run down in response to this command sequence.) If all handlers established by the image resignal the SS$_DEBUG exception, the boot handler eventually gains control.

The handler's response to an SS$_DEBUG signal is to map the debugger (if it is not already mapped) and transfer control to it. Note that an image that was neither linked nor run with the debugger can still be debugged, albeit without a debug symbol table, if the image reaches some undesirable state, such as an infinite loop.

Another function of the condition handler is to field any error conditions (where the severity level is WARNING, ERROR, or SEVERE) and pass them on to the traceback facility. If an image dump was requested, the handler dispatches to EXE$IMGDMP_MERGE (see Chapter *Condition Handling*) to create an image dump. When EXE$IMGDMP_MERGE returns, the handler maps the traceback facility, denoted by the logical name LIB$TRACE, into P0 space. If the condition has a severity level of either SUCCESS or INFO, the handler merely resignals it. The condition is then handled by the catch-all condition handler established by either EXE$PROCSTRT or the CLI that called the image.

## 8.6 IMAGE EXIT

When an image has completed its work, it passes control back to the executive, either by requesting the $EXIT system service or by returning to its caller, which requests the $EXIT system service. $EXIT calls whatever exit handlers have been declared by the image and then requests the Delete Process ($DELPRC) system service.

### 8.6.1 Exit Handlers and Related System Services

An exit handler is an optional, user-declared procedure that performs image cleanup. To establish an exit handler, an image builds a data structure called an exit control block and passes its address to the $DCLEXH system service. The access mode from which the service is requested is the mode in which the exit handler is to execute. Exit handlers can be declared for user, supervisor, and executive access modes.

An exit control block contains the procedure value of the exit handler and its arguments. The exit handler's first argument is the address of a longword to receive the final image status. The declarer of the exit handler defines any additional arguments and their use. An exit control block also contains a forward link field. This field contains the address of the next exit control block or, if there is none, zero. The $DCLEXH system service links all the exit control blocks for an access mode. Each list is ordered with the most recently declared exit handlers' control blocks first.

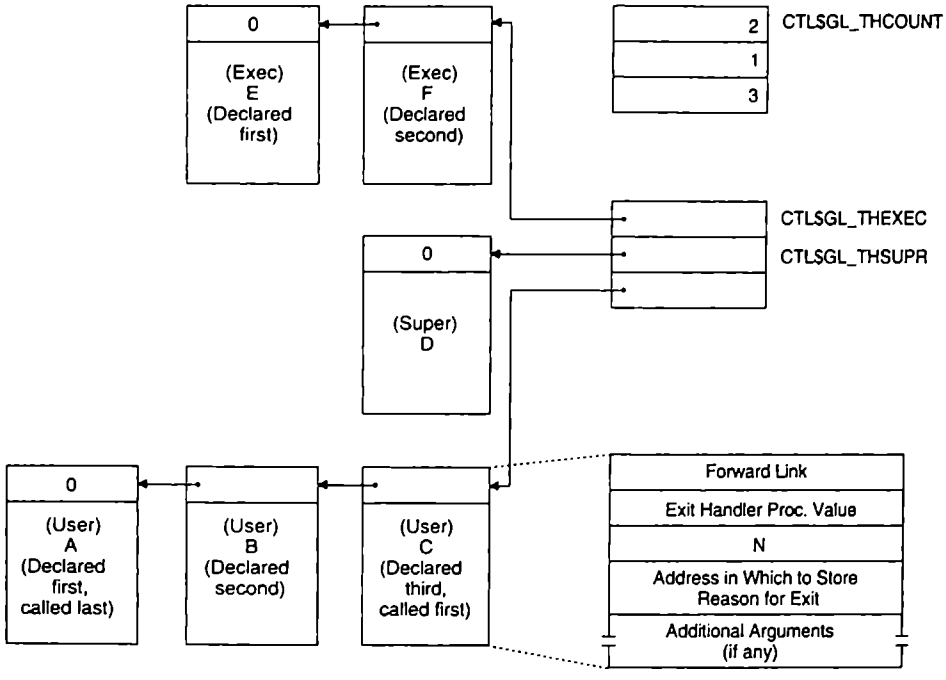



**Figure 8.27**
Sample Exit Handler Lists

The exit handler listheads are in a three-longword array. Another three-longword array contains the number of exit control blocks in each list. Each array is indexed by access mode. Figure 8.27 shows these arrays and exit control blocks.

Kernel threads within a multithreaded process synchronize access to the exit handler listheads and exit control blocks by acquiring a mutex whose address is stored in CTL$GQ_EXH_MUTEX. Both arrays are in P1 space and modifiable only from executive and kernel mode. Exit control blocks, however, are defined by the image in the process-private address space that it controls. Therefore, the system services that access these lists must exercise particular care. An exit control block corrupted through program error could destroy the integrity of its list.

When inserting or removing an exit control block, for example, each system service must test the accessibility of affected forward links. The count array is used to prevent infinite loops that might otherwise result from multiple declarations of the same exit control block.

Two system services other than $DCLEXH access exit control blocks: Cancel Exit Handler ($CANEXH) and $EXIT (see Section 8.6.2). An image requests the $CANEXH system service to delete a particular exit control block or all those for one access mode.

The $DCLEXH and $CANEXH system service procedures are EXE$DCLEXH and EXE$CANEXH, both in module SYSDCLEXH. Both execute in kernel mode.

### 8.6.2 Flow of the $EXIT System Service

The $EXIT system service procedure, EXE$EXIT in module SYSEXIT, exists as a caller's mode jacket for the $EXIT_INT system service procedure, EXE$EXIT_INT in module SYSEXIT. EXE$EXIT checks whether the process is enabled for multithreading and, if not, merely requests $EXIT_INT to call exit handlers and clean up. Otherwise, for a multithreaded process, EXE$EXIT must return to a single-threaded environment before requesting $EXIT_INT.

If executing in an inner access mode in the initial kernel thread's context, EXE$EXIT requests the $END_MULTITHREAD system service and then requests $EXIT_INT. Otherwise it queues a caller's mode asynchronous system trap (AST) to the initial kernel thread, then deletes itself by requesting the $DELETE_KERNEL_THREAD system service. The AST routine will request $END_MULTITHREAD and $EXIT_INT.

However, for a multithreaded process requesting $EXIT in user mode, EXE$EXIT requests the user mode thread manager at its EXIT upcall routine. If the thread manager returns, this thread can safely request $EXIT_INT.

If no thread manager is registered or if the thread manager has not declared an exit upcall, EXE$EXIT checks CTL$GQ_EXIT_THREAD. A nonzero value is the internal process identifier (IPID) of a thread already exiting. If no other exit is in progress, EXE$EXIT requests $EXIT_INT. Otherwise it deletes itself, or hibernates if it is the initial kernel thread. Chapter 5 provides more information about kernel threads, user mode thread manager upcalls, and the $END_MULTITHREAD system service.

EXE$EXIT_INT runs initially in kernel mode. It also executes in outer modes, calling exit handlers. It is called with a single argument, the final status of the image. It stores the status in CTL$GL_FINALSTS, from which it can be copied for image or process accounting. EXE$EXIT_INT clears the force exit pending flag, PCB$V_FORCPEN in PCB$L_STS.

If EXE$EXIT_INT was requested from kernel mode, it requests the $DELPRC system service, and the process is deleted. If EXE$EXIT_INT was requested from any other access mode, it examines the exit handler listheads (see Figure 8.27). It begins with the one for the mode from which it was requested and proceeds to those of inner (more privileged) access modes.

If EXE$EXIT_INT was requested from user mode, it elevates interrupt priority level (IPL) while it is accessing the listhead and count to prevent AST delivery, specifically the delivery of an AST enabling multithreading. If multithreading is enabled, it synchronizes its access to the listhead by acquiring the exit handler mutex. If EXE$EXIT_INT finds a nonzero listhead, it saves the listhead contents and the number of exit control blocks in the list, and clears both the listhead and the count longwords. EXE$EXIT_INT then restores the integer registers saved at entry to itself and calls EXE$REI_

INIT_STACK_ALT, in module REI_ROUTINES, to clean up the kernel stack, return to the outer access mode from which the service was requested, and enter EXIT_CALLBACK, a routine local to SYSEXIT.

Running in the outer mode, EXIT_CALLBACK removes the first exit control block from the list and saves the address of the next handler, final image status, and count of remaining handlers on the stack. It writes the final image status to the address specified in the exit control block and calls the exit handler. When (if) that handler returns, EXIT_CALLBACK calls the next handler in the list. This continues until the list is exhausted or until EXIT_CALLBACK has exhausted the count of exit handlers.

Once all the exit handlers for a given access mode have been called, EXIT_CALLBACK must return to a more privileged access mode. It clears the frame pointer (FP) register to terminate the call chain for that access mode. For a user mode thread with multithreading enabled, EXIT_CALLBACK disables user mode ASTs and sets PKTA$M_TM_ACTIVE, in PKTA$Q_TM_FLAGS, to avoid another upcall by EXE$EXIT to the user mode thread manager, if any. EXIT_CALLBACK then changes access mode by requesting the $EXIT system service. If none of the exit handlers in the list just processed has done anything extraordinary (such as declaring another exit handler), then the list for that mode is still empty and EXE$EXIT_INT proceeds to the next inner access mode in its search for more exit handlers.

When EXE$EXIT_INT reaches the executive mode exit handler list, if the list is empty, EXE$EXIT_INT tests whether deletion of this process is already in progress (PCB$V_DELPEN set in PCB$L_STS). If not, it requests the $DELPRC system service.

Otherwise, it makes several checks to determine whether, even in the absence of executive mode exit handlers, it should enter executive mode:

1. It tests and sets bit PCB$V_ERDACT in PCB$L_STS to see whether executive mode rundown routines have already been called.
2. If they have not, it tests whether any executive mode rundown routines have been established, either by the process or by the system.
3. If no rundown routines have been established, it also checks whether any executive mode process event handler has been declared to enable a subsystem, for example, the Portable Operating System Interface (POSIX) subsystem, to receive notification of image rundown.

If there is no reason to enter executive mode, EXE$EXIT_INT requests the $DELPRC system service. Otherwise, it enters executive mode as previously described so that it can call rundown routines, a process event handler, or both, from executive mode. EXIT_CALLBACK runs.

After calling executive mode exit handlers, if any, EXIT_CALLBACK tests whether any executive mode rundown routines have been established, either by the process or the system. If so, it calls each of them.

EXIT_CALLBACK then calls EXE$PRCEVT_DISPATCHER, in module LAYERED_SUBSYSTEM_ROUTINES, to inform any subsystem that requested notification of image rundown. EXIT_CALLBACK then requests the $EXIT system service.

When EXE$EXIT_INT reaches kernel mode, that is, when it has called all existing handlers, it requests $DELPRC to delete the process.

### 8.6.3 Example of Exit Handler List Processing

To illustrate the processing of exit handlers, suppose that a process has its exit handler lists set up as shown in Figure 8.27. When the image requests the $EXIT system service from user mode, EXE$EXIT_INT takes the following steps:

1. EXE$EXIT_INT finds a nonzero listhead for user mode exit control blocks. The listhead points to the exit control block for procedure C, the most recently declared user mode exit handler.
2. EXE$EXIT_INT stores this address in R0 and clears the listhead. It returns to user mode and enters EXIT_CALLBACK, which calls procedure C. When C returns, EXIT_CALLBACK calls procedure B and finally procedure A. When A returns, EXIT_CALLBACK determines that the user mode list is exhausted (because the forward pointer in the last exit handler is zero). EXIT_CALLBACK, running in user mode, requests the $EXIT system service.
3. As in step 1, the search for exit handlers begins with user mode, but this list is now empty. EXE$EXIT_INT continues with the supervisor mode list, which has the single exit control block for handler D. The supervisor listhead is cleared, access mode is changed to supervisor, and EXIT_CALLBACK calls procedure D. When D returns, EXIT_CALLBACK again requests the $EXIT system service, this time from supervisor mode.
4. Now the search for exit handlers begins with supervisor mode, whose list is empty. The list for executive mode contains two exit handlers, F and E, which are called from executive mode. When they return, the $EXIT system service is again requested, this time from executive access mode. The search that now begins with the executive mode listhead fails and the process is deleted.

The logic illustrated here shows how a process can prevent image exit through the use of exit handlers. Suppose EXE$EXIT_INT called a supervisor mode handler that redeclared itself. When EXIT_CALLBACK exhausted the exit handler list and requested the $EXIT system service again, the handler would be back on the supervisor mode exit handler list and would be reentered to redeclare itself again.

In fact, this use of exit handlers is just the mechanism employed by the DCL CLI to allow multiple images to execute, one after another, in the same process. This mechanism is discussed in more detail in Chapter 11.

Note that an exit handler that is declared later (which implies that it will be called earlier) can prevent previously declared handlers for the same access mode from even being called by simply requesting the $EXIT system service. In the previous example, procedure C could prevent exit handlers B and A from being called by requesting $EXIT itself.

### 8.6.4 RMS Exit Handler

The executive mode RMS exit handler, EXE$RMSEXH, is established by EXE$PROCSTRT. Its main purpose is to request the SYS$RMSRUNDWN system service, which performs an orderly cleanup of outstanding RMS activities by halting some I/O activity, releasing record and other locks, ensuring that disk writes which may have been pending are flushed to disk and not lost, and closing open files. SYS$RMSRUNDWN handles three classes of file:

- Indirect process-permanent files opened by the command interpreter (e.g., SYS$INPUT)
- Direct process-permanent files opened from DCL with a DCL OPEN command
- Image files

During $EXIT processing, SYS$RMSRUNDWN is called to run down and close image files and run down I/O for direct and indirect process-permanent files. During $DELPRC processing, SYS$RMSRUNDWN is called to run down and close both image and process-permanent files, aborting RMS I/O (see Chapter 10).

SYS$RMSRUNDWN performs the following steps during $EXIT processing:

1. If any record stream is associated with an open file connected to an RMS run-time unit journal file, SYS$RMSRUNDWN checks whether the stream is also associated with an open DECdtm transaction. If so, the transaction is aborted by calling SYS$ABORT_TRANSW. If the transaction is already either committing or aborting, SYS$RMSRUNDWN waits until the end-transaction activity completes.
2. It cancels outstanding I/O requests and waits for I/O activity to complete on indirect process-permanent files.
3. It allows outstanding disk I/O requests to complete, releases all record locks retained by each record stream, and requests a $CLOSE for image I/O files opened by the user.
4. It cancels outstanding I/O requests and waits for I/O activity to complete on direct process-permanent files.

5. If any open files are associated with RMS run-time unit journal files, the journal files are closed and then deleted.
6. It resets the tables of internal file access and record access blocks. Other internal RMS data structures are released.
7. If any stream is associated with a network logical link, the link is dismantled.

## 8.7 IMAGE AND PROCESS RUNDOWN

In a process with a CLI, multiple images can execute one after another. Several steps must be taken to prevent a later image from inheriting either enhancements (such as elevated privileges) or degradations (such as a reduced working set) from a previous image. In addition, when a process is deleted, all traces of it must be eliminated from the system data structures and all reusable resources returned to the system.

The $RUNDWN system service serves both those needs. $RUNDWN is called with one argument, access mode. This argument enables $RUNDWN to distinguish between image rundown and process rundown. The service is requested with an argument of user mode by the DCL CLI to clean up between image executions. $RUNDWN is also requested from the $DELPRC system service (see Chapter 10) with an argument of kernel mode to remove traces of a process being deleted.

The $RUNDWN system service performs most of its work by requesting other system services. It first obtains the access mode of its caller. It passes the access mode argument to these services to allow them to determine how much work to do. For example, the Dequeue Lock Request ($DEQ) system service (see Chapter 4) can be requested with an access mode argument to release all locks for that access mode and all outer modes. If $RUNDWN is requested with an argument of user mode, its $DEQ request cancels only user mode locks. If $RUNDWN is requested with an argument of kernel mode, then all process locks are dequeued. In this section, the phrase "based on access mode" means "perform this operation for this access mode and all outer (less privileged) access modes."

Although most rundown actions require kernel mode, executive mode rundown routines can only be called from executive mode. Therefore, the $RUNDWN system service is actually a composite mode of caller service that can request first an executive mode form of the rundown service ($ERNDWN) and then a kernel mode form ($KRNDWN).

The $RUNDWN system service procedure, EXE$RUNDWN in module SYSRUNDWN, tests whether it has been entered in kernel mode. If not, it requests the $ERNDWN service, passing it the access mode argument. In either case, EXE$RUNDWN then requests the $KRNDWN service, passing it the access mode argument. EXE$RUNDWN then returns to its caller.

The $ERNDWN service procedure, EXE$ERNDWN in module SYSRUNDWN, takes the following steps:

1. It maximizes the previous mode with its access mode argument, as previously described.
2. It tests whether executive mode rundown routines have already been called (bit PCB$V_ERDACT set in PCB$L_STS) and, if so, returns to EXE$RUNDWN. This test prevents infinite loops triggered by errors in executive mode rundown routines.
3. It requests the Change to Kernel Mode ($CMKRNL) system service to call a kernel mode procedure to set PCB$V_ERDACT (which is in a page whose protection prohibits executive mode write access).
4. It requests the Set Resource Wait Mode ($SETRWM) system service, enabling resource wait mode to ensure that image rundown completes successfully.
5. It tests CTL$GL_USRUNDWN_EXEC to see whether any executive mode rundown routines have been established. It calls each that has, passing it the access mode argument.

   It tests EXE$GL_USRUNDWN_EXEC to see if any systemwide routines have been established and, if so, calls each with the access mode argument.
6. It then calls EXE$PRCEVT_DISPATCHER, in module LAYERED_SUBSYSTEM_ROUTINES, to notify any subsystem that requested notification of image rundown.
7. It requests the $SETRWM service to restore the previous setting and returns to its requestor, EXE$RUNDWN.

The $KRNDWN service procedure, EXE$KRNDWN in module SYSRUNDWN, takes the following steps:

1. It maximizes the previous mode with its access mode argument.
2. It clears any previously requested powerfail AST and returns AST quota to the process.
3. It requests the $SETRWM system service, enabling resource wait mode.
4. EXE$KRNDWN calls any per-process or systemwide kernel mode rundown routines. Such a routine might perform cleanup for user-written system services.
5. It then calls EXE$PRCEVT_DISPATCHER, in module LAYERED_SUBSYSTEM_ROUTINES, to notify any subsystem that requested notification of image rundown.
6. It calls the License Management Facility rundown routine to release any license units granted to the exiting image or process.
7. After restoring the saved user capabilities, if necessary, EXE$KRNDWN resets the process's current CPU capability and affinity requirements to

their permanent values. If implicit affinity is enabled and the last selection is no longer in the permanent affinity CPU set, EXE$KRNDWN forces the choice of a new target CPU. It requests a scheduling interrupt if the restored affinity or capability requirements do not allow the process to execute on the current CPU. Chapter 2 explains capability and affinity requirements.

8. EXE$KRNDWN calls EXE$FANDLE_RUNDOWN, in module SYSFASTIO, to clean up the process's fandle vector. A fandle, or Fast-IO handle, identifies a set of preallocated resources used by an I/O request. EXE$FANDLE_RUNDOWN must wait for outstanding I/O requests to complete, deallocate the associated resources, and restore user quotas.
9. If image accounting is enabled, an image deletion message is written to the accounting log file. EXE$KRNDWN clears CTL$GQ_ISTART to indicate no image is active.
10. EXE$KRNDWN increments the image counter, PHD$L_IMGCNT. Use of this counter can prevent the delivery of ASTs to an image that has exited. The use of this synchronization technique in the operation of the Get Job/Process Information ($GETJPI) system service is described in Chapter 6.
11. It clears the image privilege mask, PHD$Q_IMAGPRIV.
12. It resets the activated privileged library dispatch vector (see Chapter *System Service Dispatching*) to remove user-written system services and image-temporary rundown routines.
13. It clears PCB$V_ERDACT in PCB$L_STS for later reuse.
14. Unless process-specific user mode alignment trap reporting has been enabled (see Chapter *Condition Handling*), EXE$KRNDWN disables alignment trap reporting.
15. It clears CTL$GL_NATIVE_TO_TIE, used by the $NATIVE_TO_TRANSLATED system service.
16. It calls EXE$WRITE_THREAD_UNQ, in module SYSRUNDWN, to clear thread-specific context.
17. EXE$KRNDWN requests the Set Page Fault Monitoring ($SETPFM) system service to disable any monitoring of process page faults.
18. EXE$KRNDWN searches the channel control block table for channels to deassign. It compares the access mode of each assigned channel to that of the rundown. For each channel assigned in the same or an outer mode, EXE$KRNDWN requests the Deassign Channel ($DASSGN) system service. The deassign completes unless the channel has an open file. The access mode comparison prevents process-permanent files from being closed when an image is being run down ($RUNDWN from user mode). Other channels that are not deassigned at this stage of image rundown include the image file and any other file that is mapped to a range of virtual addresses.

If the channel's assigned mode is more privileged, EXE$KRNDWN makes an additional check of the flag CCB$V_IMGTMP to see whether the channel is associated with the Breakthrough ($BRKTHRU) system service. If it is, EXE$KRNDWN deassigns the channel so that broadcast operations are aborted at image exit.

19. The rights database identifier table is deallocated to the P1 process allocation region.
20. EXE$KRNDWN requests the Cancel Timer ($CANTIM) and the Cancel Wakeup ($CANWAK) system services to cancel any requests made from this and outer access modes.
21. If any process has established a debug connection to this process, it requires notification of the termination. In such a connection, established by the Process Termination Notification ($PRCTERM) system service, each connected process acquires a concurrent read lock on the system resource $PRCNAME + process ID, specifying a blocking AST address and context block. EXE$KRNDWN now requests an exclusive lock on that resource, causing any blocking AST to be delivered. The AST routine, running in the context of the connected process, can deallocate resources and perform any desired cleanup.
22. It requests the $DEQ service to release all process-owned locks for this and outer access modes.
23. The connected process in a debug connection maintains a queue of context blocks at CTL$GQ_PRCTERM_LISTHEAD. If this process has such a list, EXE$KRNDWN raises IPL to 2 to prevent AST delivery and releases the associated locks. (Since these are system-owned locks rather than process-owned locks, they are not released in the previous step.) After lowering IPL, it deallocates the context blocks.
24. EXE$KRNDWN calls NSA$DEACCESS_AUDIT, in module SECURITY_AUDITING, with the group global section and system global section object class blocks to flush all pending deaccess audits.
25. EXE$KRNDWN calls MMG$IMGRESET, in module PHDUTL, to reset the image pages. MMG$IMGRESET performs the image cleanup associated with memory management:

    a. MMG$IMGRESET calls MMG$EXTRADYNWS, in module SYSADJWSL, to recalculate PHD$L_EXTDYNWS, the number of dynamic working set list entries (WSLEs). Working sets are described in Chapter *Working Set List Dynamics*.
    b. It calls RM$RESET, in module RMSRESET, to reset the image I/O segment.
    c. It returns memory management working set peak checking to its previous state.
    d. At IPL 2, MMG$IMGRESET calls EXE$PSCAN_IMGRESET, in module PROCESS_SCAN, to remove and deallocate process scan blocks,

restoring the context of the Process Scan ($PROCESS_SCAN) system service.

e. It calls MMG$DELETE_BUFOBJ, in module SYSLKWSET, to delete any buffer objects still allocated by the process except those in user-defined, process-permanent regions.

f. After lowering IPL, it releases all IMCBs that describe currently mapped images and places them into the IMCB lookaside list. Each IMCB that describes an image installed /RESIDENT also has an associated KFE and KFERES. MMG$IMGRESET deallocates the P1 copy of the KFERES and decrements the reference count in the associated KFE. If the count goes to zero and deletion is pending for the KFE, MMG$IMGRESET deallocates the huge page slice that had been occupied by the image and deallocates the KFE and, if necessary, the KFERES.

   If any IMCBs remain in the work list, MMG$IMGRESET places them into the IMCB lookaside list as well.

g. Setting IPL to 2, it locates each process-private region, beginning with P0 space. For regions other than those marked permanent, including P0, P1, and P2, MMG$IMGRESET deletes the virtual address space and the region descriptor. It then deletes the virtual pages of the P0 region, the P2 region, and the nonpermanent parts of P1 space. This includes the user stack and an optional enlarged image I/O segment (see Figure 8.28), plus any expansions to P1 space (at smaller virtual addresses than the user stack) and DEBUG dynamic memory. It also deletes the P1 image region, where installed images with shareable address data are mapped.

   All of P0 space is deleted. This frees the main image file and all other image files currently mapped. Physical pages are released, and blocks in the page files assigned to the process are deallocated.

h. The working set list is reset to its default value, undoing any previous expansion or contraction performed by the Adjust Working Set Limit ($ADJWSL) system service. Working set size changes are described in Chapter *Working Set List Dynamics*.

i. At IPL 2, MMG$IMGRESET calls MMG$SECTBLRST, which compresses the process section table.

j. The process privilege masks in the PHD and the process control block (PCB) are reset to their permanent value, found at location CTL$GQ_PROCPRIV. This step eliminates any privilege enhancements to the process resulting from the execution of an image installed with privilege. Section 8.8 describes the various privilege masks.

k. The global location CTL$GL_IMGHDRBF is cleared to indicate that no image is active.

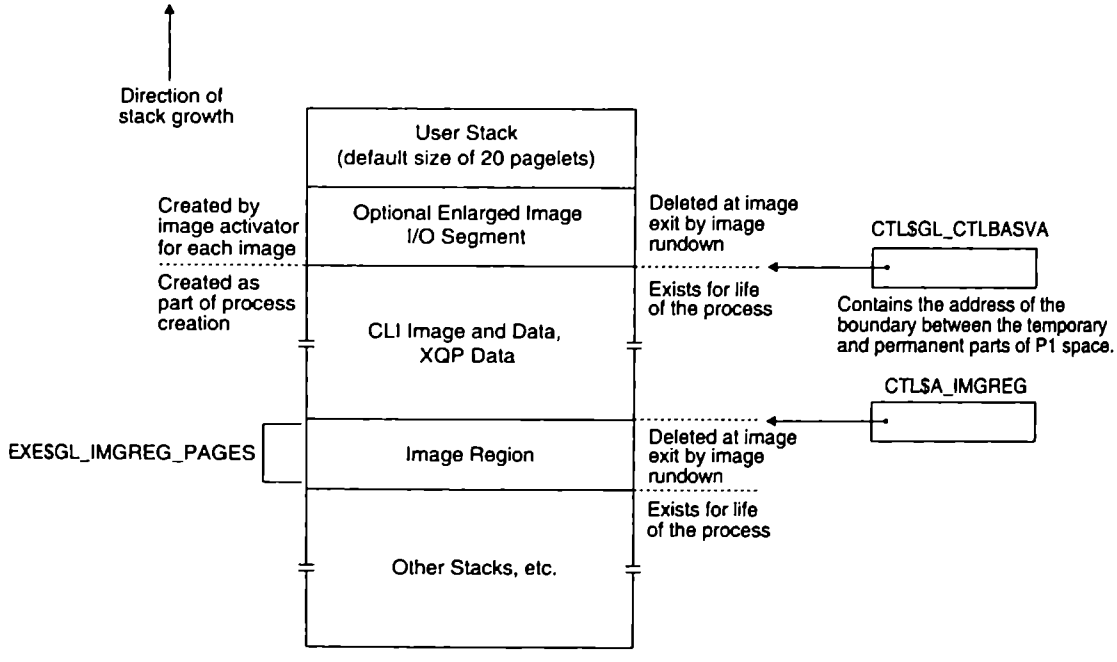

Direction of stack growth
User Stack (default size of 20 pagelets)
Created by image activator for each image
Optional Enlarged Image I/O Segment
Deleted at image exit by image rundown
CTL$GL_CTLBASVA
Created as part of process creation
Exists for life of the process
Contains the address of the boundary between the temporary and permanent parts of P1 space.
CLI Image and Data, XQP Data
CTL$A_IMGREG
EXESGL_IMGREG_PAGES
Image Region
Deleted at image exit by image rundown
Exists for life of the process
Other Stacks, etc.

**Figure 8.28**
Low-Address End of P1 Space That Is Deleted at Image Exit

l. If the process was the last accessor of a global section, releasing the process address space may make the global section deletable. If so, the global sections are deleted under the protection of the global section mutex. The associated WCB is released as well.

m. The pointer to the end of the active working set list, PHD$L_WSLAST, is reset to the end of the minimum working set list.

26. The channel deassignment loop performed in step 18 is executed again. However, because the image file and other mapped files have now been dissociated from virtual address space, the channels associated with those files will also be deassigned. As in step 18, this deassignment is based on access mode, so that process-permanent files are unaffected by image rundown.
27. EXE$KRNDWN requests the Deallocate Device ($DALLOC) system service to deallocate devices allocated from this and outer access modes.
28. It requests RSDM$RUNDOWN, in module RESOURCE_DOMAIN_OBJECT, to disassociate the process from system security objects and release the related resources.
29. It requests the Disassociate Common Event Flag Cluster ($DACEFC) system service to dissociate clusters 2 and 3.

30. EXE$KRNDWN acquires the SCHED spinlock, elevating IPL to IPL$_SCHED.
31. It checks the system error log mailbox queue EXE$AQ_ERLMBX and deassigns each error log mailbox belonging to this process (see Chapter *Error Handling*).
32. It removes all pending AST control blocks (ACBs) from the queues in the kernel thread block (KTB), based on access mode. If the rundown access mode is kernel, it also removes any queued special kernel mode ASTs. If user AST quota was charged for the AST, it returns the quota and deallocates the ACB to nonpaged pool unless its ACB$V_NODELETE or ACB$V_PKAST bit is set. If the ACB describes a piggyback special kernel mode AST, EXE$KRNDWN releases the SCHED spinlock, calls the AST routine, and reacquires the spinlock to continue scanning the queues. It clears the bit corresponding to the access modes of each emptied queue in KTB$L_AST_PENDING.

    It scans any inner mode queues that were not emptied for ACBs whose AST procedure is in P0 space, which has already been deleted. It removes any such ACB and processes it as the others: possibly returning AST quota, invoking a piggyback kernel AST routine, and deallocating the ACB.
33. If the rundown mode is user mode, EXE$KRNDWN removes any declaration of a user change mode handler and zeros the catch-all extension handler list (see Chapter *Condition Handling*).
34. Any exit handlers for this access mode are canceled.
35. Condition handlers declared as the primary, secondary, and last chance are eliminated for this and outer access modes.
36. The AST active bits for this and outer access modes are cleared. The AST enable bits for this and outer access modes are set.
37. System service failure exceptions are disabled for this and outer access modes.
38. EXE$KRNDWN calls SCH$RESET_ASTSR, in module ASTDEL, to calculate a new value for the process's AST summary register to reflect the change in the AST queue resulting from step 32.
39. The force exit pending (PCB$V_FORCPEN) and wake pending (PCB$V_WAKEPEN) flags in the PCB are cleared. After clearing these flags, EXE$KRNDWN releases the SCHED spinlock, lowering IPL to 0.
40. A protected subsystem, for example, the POSIX subsystem, stores its image rights list segment in CTL$GL_SUBSYS_RIGHTS while inactive, and in the access rights block (ARB) while active. EXE$KRNDWN locates the image rights list segment, if any, deallocates the image rights list segment at IPL 2, and clears the ARB and P1 pointers.
41. EXE$KRNDWN deletes all process logical names based on access mode. At image exit, all user mode logical names are deleted. At process deletion, all process logical names are deleted.

42. EXE$KRNDWN resets any P0 extension made to the process allocation region (see Chapter *Pool Management*).
43. Resource wait mode is returned to its previous state, normal completion status is set, and control is returned to its requestor, EXE$RUNDWN.

### 8.7.1 Inner Mode Rundown Routines

The inner mode rundown mechanism enables an executive or kernel mode routine to ensure that it gains control at image rundown or process deletion to perform a specific rundown task. Typically a rundown routine is associated with some other inner mode routine that has acquired system resources or changed process or system state on behalf of the process. A rundown routine differs from an exit handler in the following ways:

- Kernel mode rundown routines can be declared, whereas there are no kernel mode exit handlers.
- Exit handlers are not guaranteed to be called at image rundown. When, for example, a process is the target of a $DELPRC request, its current image is run down but no exit handlers are called. In contrast, kernel mode rundown routines are always called; executive mode rundown routines are called unless the process deletes itself from executive or kernel mode.

Typically rundown routines are part of a privileged shareable image. The privileged shareable image is built with a PLV that lists any executive and kernel mode rundown routines in the image, in addition to any executive and kernel mode user-written system services. Such an image must be installed with the /PROTECTED qualifier. Consult the *OpenVMS Programming Concepts Manual* for information on building a privileged shareable image.

When an image with a PLV is activated, the image activator examines the PLV and adds entries for the declared rundown routines to a P1 space structure called the activated privileged library dispatch vector (see Chapter *System Service Dispatching*).

Figure 8.29 shows the fields in this structure that relate to rundown routines. Beginning at offset APLD$PS_EXEC_RUNDOWN_VECTOR is space to accommodate the procedure values of 42 executive mode rundown routines. The procedure value of the first rundown routine is stored at offset APLD$PS_EXEC_RUNDOWN_VECTOR; the procedure value for the next rundown routine is stored in the next longword. A longword containing zero indicates the end of the list. APLD$L_EXEC_RUNDOWN_INDEX is the longword index of the next available longword in the list.

During process creation, the address of this list is recorded in CTL$GL_USRUNDWN_EXEC.

Rundown routines are categorized as image-temporary or permanent. The addresses of image-temporary routines are cleared at image rundown. The

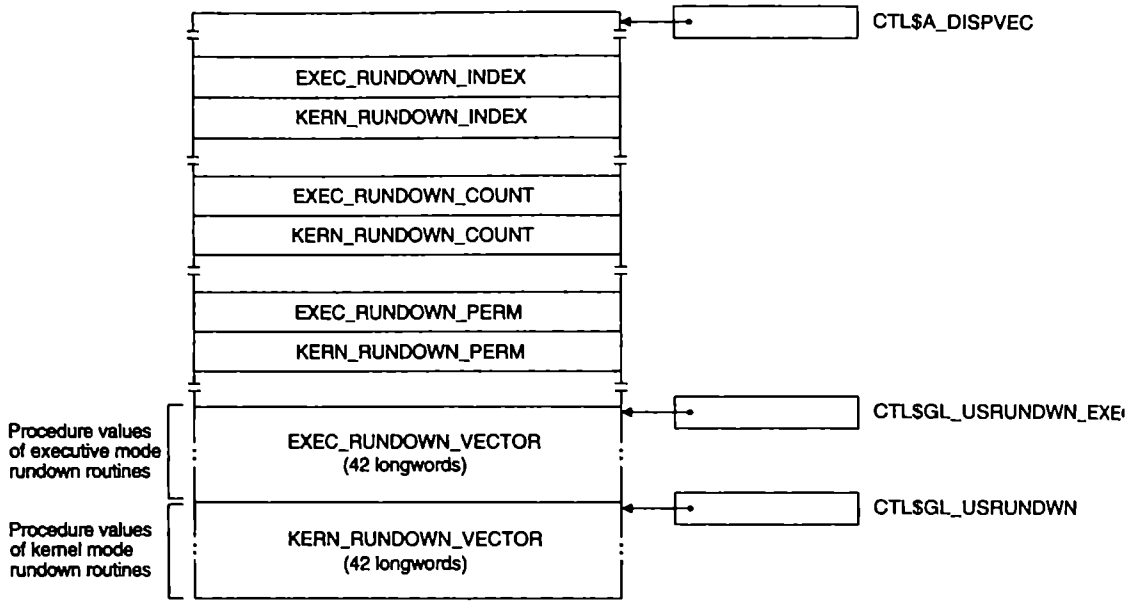



**Figure 8.29**
Rundown Vectors

addresses of permanent routines persist across image rundowns and activations. All image-temporary routines are entered from the beginning of the list, followed by permanent routines. The number of image-temporary executive mode rundown routines in the list is contained in APLD$L_EXEC_RUNDOWN_COUNT, and the number of permanent rundown routines in APLD$L_EXEC_RUNDOWN_PERM.

Rundown routines entered by the image activator when it processes a protected image are image-temporary. Kernel mode code can enter a rundown routine, either kernel or executive mode and either image-temporary or permanent, by directly calling IMG$ADD_PRIVILEGED_VECTOR_ENTRY. Use of this routine is reserved to Digital. Any other use is unsupported.

The activated privileged library dispatch vector structure contains a similar list and fields for kernel mode rundown routines. CTL$GL_USRUNDWN contains the address of the beginning of the kernel mode rundown routine list, offset APLD$PS_KERN_RUNDOWN_VECTOR.

Rundown routines can be called from any one of several OpenVMS system services:

- $EXIT_INT (see Section 8.6.2)
- $ERNDWN and $KRNDWN (see Section 8.7)
- $DELPRC (see Chapter 10)

The goals are that rundown routines are called once, regardless of how the current image is rundown, and that they are called no more than once.

Bit PCB$V_ERDACT in PCB$L_STS, when set, indicates either that executive mode rundown routines have been called already or that the executive is about to enter executive mode to call them. It helps ensure that executive mode routines are called no more than once.

After the $KRNDWN system service calls kernel mode rundown routines, it clears the activated privileged library dispatch vector of all image-temporary rundown routines, both executive and kernel mode.

## 8.8 PROCESS PRIVILEGES

The executive prevents unauthorized use of the system through process privileges. One or more of these privileges are required to request particular system services, execute certain commands, or use privileged utilities. The use of privilege is audited by the system.

### 8.8.1 Process Privilege Masks

A process has three sets of privileges available to it: privileges available while executing a particular image, privileges available to the current process context, and privileges from which the process can selectively alter its current context. Each set of privileges is represented by a quadword bit mask. A set bit means the process has the privilege corresponding to that bit.

The executive maintains a number of privilege masks for processes and images. Table 8.2 summarizes the use of the masks.

PCB$Q_PRIV exists in the ARB, which is a part of the PCB. It is also referenced by the symbol ARB$Q_PRIV. PCB$Q_PRIV contains the working privilege mask. This mask is checked by most system services that require privilege and by the file system. It is the default privilege mask checked by the $CHECK_PRIVILEGE system service. At image activation, the mask is initialized to the combination of the privileges of the image and the privileges of the current process context. It can be altered by the Set Privileges ($SETPRV) system service, either during image execution or from DCL level. It is reset at image rundown to the current process privileges.

The other image-specific privilege mask is PHD$Q_PRIVMSK in the process header. It is a duplicate of the privilege mask in the ARB and is altered in the same manner. Some older system services reference this mask rather than ARB$Q_PRIV.

Current process privileges (also called process-permanent privileges) are stored in the P1 pointer area at global location CTL$GQ_PROCPRIV. This mask is initialized at process creation from the user authorization file (UAF) default privilege mask; from the privilege mask argument passed to the Create Process ($CREPRC) system service; or, for a subprocess, from the creator's current privilege mask. It can be altered by the $SETPRV system service, either during image execution or from DCL level. Its contents are copied to the working privilege mask at image rundown.

Table 8.2 Process Privilege Masks

| *Symbolic Name* | *Use of This Mask* | *Modified by* | *Referenced by* |
|---|---|---|---|
| PCB$Q_PRIV, ARB$Q_PRIV | Working privilege mask | EXE$PROCSTRT, LOGINOUT, $SETPRV, Image activator, MMG$IMGRESET | Device drivers, XQP, ACPs, system services requiring privilege |
| PHD$Q_PRIVMSK | Duplicate of ARB mask | Same as PCB$Q_PRIV | Some system services requiring privilege |
| CTL$GQ_PROCPRIV | Records permanently enabled privileges | EXE$PROCSTRT, LOGINOUT, $SETPRV | Image activator, SET/SHOW commands, MMG$IMGRESET |
| PHD$Q_AUTHPRIV | Records privileges from authorization file | EXE$PROCSTRT, LOGINOUT | $SETPRV, $GETJPI |
| PHD$Q_IMAGPRIV | Records privileges of installed image | Image activator, $RUNDWN | $SETPRV, LOGINOUT, $GETJPI |
| UAF$Q_PRIV | Records privileges in authorization file | AUTHORIZE | LOGINOUT |
| UAF$Q_DEF_PRIV | Records default privileges in authorization file | AUTHORIZE | LOGINOUT |
| KFE$Q_PROCPRIV | Records privileges with which an image is installed | Install utility | Image activator |

The authorized privilege mask, PHD$Q_AUTHPRIV, does not change over the life of the process. It allows a process to remove a privilege from its current privilege mask with the $SETPRV system service and to later regain that privilege. The authorized privilege mask is initialized at process creation from the UAF privilege mask; from the privilege mask argument passed to the $CREPRC system service; or, for a subprocess, from the creator's current privilege mask.

Each UAF record contains two privilege masks: UAF$Q_DEF_PRIV and UAF$Q_PRIV. UAF$Q_DEF_PRIV contains the default privileges that LOGINOUT copies to CTL$GQ_PROCPRIV, PCB$Q_PRIV, and PHD$Q_PRIVMSK when an interactive user logs in. UAF$Q_PRIV contains the authorized privileges that LOGINOUT copies to PHD$Q_AUTHPRIV.

KFE$Q_PROCPRIV records the privileges with which a known executable image has been installed. When a process runs such an image, those privileges are temporarily granted to the process as part of the working privilege mask.

PHD$Q_IMAGPRIV contains a copy of the privileged known image's KFE$Q_PROCPRIV mask while that image is executing in the process context. This mask is used by the $SETPRV system service to allow an im-

age installed with privilege to request the $SETPRV service without losing privileges.

### 8.8.2 $SETPRV System Service

The $SETPRV system service enables a process to alter its image-specific (PCB$Q_PRIV and PHD$Q_PRIVMSK) privilege masks or its image-specific and process-permanent (CTL$GQ_PROCPRIV) privilege masks, gaining or losing privileges as a result. In addition, the service can return the previous settings of either the image-specific or process-permanent privileges, if requested.

The $SETPRV system service procedure, EXE$SETPRV in module SYSSETPRV, runs in kernel mode.

The path through EXESSETPRV that disables privileges requires no special privilege and clears the requested privilege bits in the image-specific and, optionally, the process-permanent privilege masks.

The code path that enables privileges requires the requested privilege to be already included in the mask of privileges authorized for this process, PHD$Q_AUTHPRIV. If a process tries to acquire a privilege that is not in its authorized mask, the requested privilege is still granted if any one of the following three conditions holds:

- The process has SETPRV privilege in its authorized mask. A process with this privilege can acquire any other privilege with either the $SETPRV system service or the DCL command SET PROCESS/PRIVILEGES (which requests the $SETPRV system service).
- The system service was requested from executive or kernel mode. This condition allows code running in either access mode, including user-written system services, to acquire whatever privileges it needs without regard for whether the current process has SETPRV privilege. Such procedures must disable privileges granted in this fashion as part of their return path.
- The privilege is being acquired temporarily (enabled in the two image-specific privilege masks) and is included in the mask of privileges authorized for the image, PHD$Q_IMAGPRIV, or the SETPRV privilege is included in this mask. This allows an image to acquire a privilege without permanently granting the new privilege to the process. When the image exits, image rundown copies the process-permanent mask to the image-specific masks, removing privileges acquired temporarily.

Note that the implementation of the $SETPRV system service does not return an error if a nonprivileged process attempts to add unauthorized privileges. In such a case, the service clears all unauthorized bits in the requested privilege mask, loads the modified privilege mask, and returns the alternative success status SS$_NOTALLPRIV.

## 8.9 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[INSTAL]FIXSQUISH.C
[INSTAL]INSCREATE.B32
[INSTAL]INSMAIN.B32
[INSTAL]INSRESIDENT.B32
[LIB]EIAFDEF.SDL
[LIB]EICPDEF.SDL
[LIB]EIHADEF.SDL
[LIB]EIHDDEF.SDL
[LIB]EIHIDEF.SDL
[LIB]EIHPDEF.SDL
[LIB]EIHSDEF.SDL
[LIB]EIHVNDEF.SDL
[LIB]EISDDEF.SDL
[LIB]IMCBDEF.SDL
[LIB]KFDDEF.SDL
[LIB]KFEDEF.SDL
[LIB]KFERESDEF.SDL
[LIB]KFHDEF.SDL
[LIB]KFIDEF.SDL
[LIB]KFPBDEF.SDL
[LIB]KFPDEF.SDL
[LIB]KFRHDEF.SDL
[LIB]SHLDEF.SDL
[SYS]IMGACT_SUPPORT.C
[SYS]IMGACTCTX.SDL
[SYS]IMGDECODE.B32
[SYS]IMGMAPISD.B32
[SYS]PHDUTL.MAR
[SYS]PROCSTRT.MAR
[SYS]RELOCATE_AND_FIXUP.B64
[SYS]SYSDCLEXH.MAR
[SYS]SYSEXIT.MAR
[SYS]SYSIMGACT.B32
[SYS]SYSIMGFIX.B64
[SYS]SYSIMGSTA.MAR
[SYS]SYSRUNDWN.MAR
[SYS]SYSSETPRV.MAR

# 9 Translated Image Environment

Untwisting all the chains that tie
The hidden soul of harmony.
Milton, *L'Allegro*

DECmigrate for OpenVMS Alpha Systems is a collection of tools that eases the migration of OpenVMS VAX applications to OpenVMS Alpha systems. The DECmigrate product provides the VAX Environment Software Translator (VEST) utility, which translates an OpenVMS VAX image to an OpenVMS Alpha image. This chapter briefly describes the VEST utility. Its main focus is the run-time environment in which a translated image executes and OpenVMS Alpha support of that environment.

## 9.1 OVERVIEW

During the design of the Alpha architecture, Digital considered the challenges involved in migrating existing OpenVMS VAX applications to Alpha processors. Recompiling and relinking the modules of a single image to take advantage of Alpha performance is straightforward if the modules are written in a language for which an Alpha compiler is available. Migrating a complex application consisting of many executable and shareable images is a more difficult undertaking, however, particularly if not all the sources are available or if they are written in a language for which no Alpha compiler is currently available.

To facilitate more difficult migrations, Digital provides software that can translate an OpenVMS VAX image binary file into one that can run on an Alpha system. A translated VAX image contains Alpha instructions that reproduce the behavior of the VAX image, including simulated compliance with implicit machine state, instruction side effects, and the VAX calling standard. The article "Binary Translation" in *Communications of the ACM* 36 (February 1993) or in *Digital Technical Journal* 4, no. 4 (1992) describes the advantages of binary translation over interpretation and emulation, and explains the design of the binary translator.

One design goal for the binary translator was that it perform automatic open-ended translation of almost all user mode images. (Section 9.3.3 describes image characteristics that prevent successful translation.) In open-ended translation, not all instructions are necessarily found and translated during one translation; some may be discovered, modified, or created during image execution. By default, during image execution, a feedback file is generated, which identifies newly found entry points and code, and which can be input to a later translation.

Another critical design goal for the binary translator was optional exact emulation of VAX architecture details such as precise arithmetic traps and certain atomic memory accesses.

The result of this design is the VEST utility, the major part of the DECmigrate product. The VEST utility translates OpenVMS VAX executable and shareable images into OpenVMS Alpha images.

Once translated, an image requires run-time support to make transparent the differences in system environment between OpenVMS VAX and OpenVMS Alpha systems, for example, differences in calling standards, architectural features, and exception handling. Run-time support for translated images is bundled with the OpenVMS Alpha system. In addition to support within the executive, it consists of the following components:

- The jacket system services, $NATIVE_TO_TRANSLATED and $TRANSLATED_TO_NATIVE, which enable calls between native and translated images (see Section 9.4)
- TIE$SHARE.EXE, the shareable image that manages the emulated VAX state required by the translated image, interprets VAX code that was not translated, and emulates D floating-point and H floating-point instructions and certain other frequently used untranslatable instructions such as `REI`, `RET`, and `MOVC3`.
- TIE$EMULAT_TV.EXE, a shareable image created by translating existing VAX emulation routines, with procedures for emulating less frequently used complex VAX instructions such as `POLYD`
- Translated versions of various run-time libraries (RTLs) shipped with the OpenVMS VAX system

A translated image consists of the original VAX image, native instructions generated by translating VAX instructions, calls to procedures in TIE$SHARE, and if necessary, calls to procedures in TIE$EMULAT_TV.

A translated image runs in an OpenVMS Alpha environment. Its code can call native procedures and request native services. Native images can call procedures in translated shareable images, and the executive can call translated procedures, for example, to deliver an asynchronous system trap (AST). Jacketing routines provided by the executive facilitate cross-architecture calls. Native and translated code share the same stack, with VAX call frames and native procedure frames interleaved by TIE$SHARE. In this chapter, *native* means native within the Alpha environment.

## 9.2 EMULATING THE VAX ENVIRONMENT

The environment in which user mode instructions execute on a VAX consists of the following:

- Memory
- General registers

Table 9.1 Register Use in a Translated Image

| *VAX Register* | *Use* | *Alpha Register* |
|---|---|---|
| R0–R14 | Integer arithmetic, address calculation | R0–R14 |
| R0–R12 | Floating-point arithmetic | F0–F12 |
| R12 | VAX argument pointer | R12 |
| R13 | VAX frame pointer | R13 |
| R14 | VAX stack pointer | R14 |
| R15 | VAX PC-relative addressing | R15 |
| n/a | Alpha procedure calls; temporary register | R16–R21 |
| Processor status | Condition codes and exception enable bits | R22 |
| N and Z condition codes | | R23 |
| n/a | Alpha scratch | R24 |
| n/a | Alpha procedure calls; temporary register | R25 |
| n/a | Alpha procedure calls | R26, R27 |
| n/a | Alpha scratch | R28 |
| n/a | Alpha frame pointer | R29 |
| n/a | Alpha stack pointer | R30 |
| n/a | Alpha sink and zero source | R31 |

- Condition codes
- Stack
- Program counter (PC)

The VEST utility and the OpenVMS Alpha system cooperate to emulate a VAX environment for the execution of translated VAX images. A translated image contains a copy of the VAX image with instructions and data at the same locations as in the VAX image. The native instructions generated in translation by VEST reference statically allocated data at the same locations as the VAX instructions do.

Transforming VAX register use into Alpha register use is straightforward because there are more Alpha registers than VAX registers. VAX register contents are sign-extended to 64 bits. Table 9.1 shows how VAX register use is translated.

Execution of any VAX instruction potentially affects four condition codes, which can be tested by a subsequent branch instruction. To improve performance, translated code calculates condition codes only when necessary. When emulated, all four condition codes are recorded in R22. To facilitate testing whether results are negative, zero, or positive, the N and Z condition codes are recorded in R23. If the N condition code is set, the result is negative. If the Z condition code is set, the result is zero. If both are clear, the result is positive.

Translated VAX code and the native routines that support it share the user mode stack. Translated code refers to the stack using R14, the VAX stack

pointer (SP), and R13, the VAX frame pointer (FP). Section 9.6 provides more information on how frames interleave.

The VAX PC is a general register that can be used in operand addressing in addition to identifying the instruction being executed. Translated code emulates PC-relative addressing using R15. When used in this way, R15 typically contains the sum of the base address of the translated image plus $8000_{16}$ rather than the base address. This extends the range within the image that a signed displacement from R15 can reach by enabling a negative displacement.

The VAX architecture includes several features not present in the Alpha architecture. Correct execution of some translated images depends on the presence of these features. The VEST utility and TIE$SHARE must therefore emulate them.

The sections that follow describe the problematical VAX architectural features, Alpha PALcode support for VAX environment emulation, and VEST and TIE$SHARE support.

### 9.2.1 Significant VAX and Alpha Architectural Differences

The VAX architecture differs in several ways from the Alpha architecture. Areas of difference of particular significance for translation and execution of VAX images are

- D floating-point precision
- H floating-point format
- Precise arithmetic exceptions
- Form of floating-point zeros
- Access to unaligned data
- Byte granularity of memory access
- Atomic read-modify-write operations
- Ordered reads and writes to memory
- Instruction atomicity

By default VEST generates special code sequences that reproduce VAX behavior in some of these regards. For images that depend on other aspects of VAX behavior, VEST optionally generates special code sequences that reproduce it. Some sequences require TIE$SHARE support as well.

Because of limited D floating-point hardware support, by default VEST converts any D floating-point operands to G floating-point operands and generates G floating-point instructions. This conversion loses three bits of precision in the fractional part of the number and trades that precision for improved performance. The VEST qualifier /FLOAT=D56_FLOAT overrides the default. In response to that qualifier, VEST generates calls to TIE$SHARE software emulation routines for the D floating-point operations. Use of this qualifier trades performance for arithmetic precision.

The Alpha architecture does not support H floating-point arithmetic. When VEST detects its use in an image being translated, it generates calls to TIE$SHARE software emulation routines.

The VAX architecture requires that exceptions be precise; that is, if execution of an instruction triggers an exception, no instructions in the stream following the one incurring the exception may be executed. The exception PC precisely identifies the instruction that incurred the exception. In contrast, the Alpha architecture permits imprecise arithmetic exceptions. Under normal operation, a pipelined Alpha processor can be executing several instructions in various stages at any given instant; one or more of these can trigger an arithmetic trap. The exception PC does not precisely identify the instruction that incurred the exception. Moreover, an exception can represent more than one error. Exception summary information passed from PALcode identifies which errors occurred.

By default VEST rearranges instructions so as to promote multiple-instruction issue. A VAX image that requires precise exceptions must be translated with the qualifier /PRESERVE=(FLOAT_EXCEPTIONS,INTEGER_EXCEPTIONS). In response to that qualifier, VEST does not rearrange generated code except within the set of instructions generated for one VAX instruction, and it generates TRAPB instructions after each floating-point operation and after any integer operation that could incur an integer overflow. The TRAPB instruction allows software to guarantee that previously issued arithmetic instructions complete and trigger any pending exceptions before any instructions following the TRAPB are issued. Use of this qualifier trades performance for precision of exception information and an ability to fix up memory operands that cause floating-point overflows or underflows.

A VAX CPU interprets as zero a floating-point pattern with a zero exponent, zero sign bit, and nonzero fraction. Such a pattern is referred to as a dirty zero. In contrast, use of a dirty zero in most Alpha floating-point instructions results in an invalid-operation arithmetic exception. When such an error is reported by itself in an arithmetic exception, TIE$SHARE fixes up the operand to create a true zero (64 bits of zero) and repeats the floating-point instruction.

As described in Chapter *Synchronization Techniques*, Alpha memory references are more restricted than VAX memory references. One distinction is that Alpha memory references are limited to accessing one aligned longword or quadword. By default, in translating a VAX image, VEST assumes that longwords, quadwords, and words whose alignment it cannot determine are naturally aligned. When an instruction references unaligned data, the hardware traps to a PALcode routine that emulates the reference, breaking it into multiple aligned loads or stores. Trapping in this way decreases performance. A VAX image known to have unaligned data can be translated with the VEST qualifier /OPTIMIZE=NOALIGN. In response to that qualifier, VEST generates multiple instructions to load or store the data. Executing those

instructions in-line is faster than trapping to a PALcode routine and having it execute them.

A VAX CPU is required to read or write certain memory operands in a single operation that appears indivisible from the viewpoint of any other thread of execution on a symmetric multiprocessing (SMP) system. These operands include a byte, aligned word, and aligned longword. Certain other types of memory access must appear indivisible from the viewpoint of any other thread of execution on the same CPU. Moreover, a VAX CPU can execute an interlocked instruction to effect a read-modify-write sequence that appears atomic from the viewpoint of any other thread of execution.

The Alpha architecture has no one instruction that both reads and writes memory. It provides only that an aligned longword or aligned quadword can be read or written atomically with respect to other threads on any CPU of an SMP system. Writing a byte, word, or unaligned longword within one quadword requires a sequence of instructions. The load-locked/store-conditional mechanism, described in Chapter *Synchronization Techniques*, can be used to implement atomic reads or writes to a byte, word, or longword within a quadword. This mechanism can also be used to implement read-modify-write sequences that are atomic from the viewpoint of any other thread of execution.

The atomic move PALcode routine AMOVRM, described in Section 9.2.2, uses the load-locked/store-conditional mechanism to implement writes to multiple quadwords that are atomic from the viewpoint of other threads on the same CPU but not from the viewpoint of threads on other CPUs. This level of atomicity, however, duplicates that of the VAX architecture. The AMOVRM routine can be used, for example, to write atomically an unaligned word or longword that crosses a quadword boundary.

A program that shares memory data with another thread of execution may need to be translated with the VEST qualifier /PRESERVE=MEMORY_ATOMICITY regardless of whether the other thread is an AST, a condition handler, or a program running in another process. If the data is shared explicitly and only accessed with interlocked VAX instructions (for example, `ADAWI` or `INSQTI`), the qualifier is not needed for the VEST utility to generate atomic accesses. If the data is shared implicitly and consists of other than aligned longwords and quadwords, use of the qualifier is necessary for atomic accesses. (Chapter *Synchronization Techniques* explains how two fields in the same quadword accessed by different threads of execution are implicitly shared.) In response to this qualifier, VEST generates code to access VAX instruction memory operands (other than aligned longwords and quadwords) using the load-locked/store-conditional mechanism or the AMOVRM PALcode routine.

The VAX architecture requires that in a multiprocessor system one CPU's reads from memory appear ordered from the viewpoint of another CPU and that one CPU's writes to memory appear ordered from the viewpoint of an-

other CPU. In contrast, the Alpha architecture provides no implicit ordering. Any software that relies on read/write ordering must explicitly order the reads or writes. The fundamental ordering mechanism is the memory barrier (MB) instruction, described in Chapter *Synchronization Techniques*. By default VEST provides no ordering. A program that is part of a multiprocess application intended for an SMP system can be translated with the VEST qualifier /PRESERVE=READ_WRITE_ORDERING. In response to that qualifier, VEST limits its instruction scheduling and generates MB instructions and CALL_PAL AMOVRM to access VAX instructions' memory operands.

Section 9.2.3 discusses VAX instruction atomicity and the ways in which that aspect of VAX behavior is reproduced for translated images.

### 9.2.2 Alpha Architectural Support for Translated Images

The Alpha architecture provides the following features specifically in support of the translated image environment:

- Read and set (RS) instruction
- Read and clear (RC) instruction
- Atomic move PALcode routines AMOVRR and AMOVRM

These features are provided only for support of translated code. Any other use is unsupported.

When translating an image with the qualifier /PRESERVE=INSTRUCTION_ATOMICITY, VEST generates code that uses the RS instruction and CALL_PAL AMOVRM to preserve VAX-style instruction atomicity (see Section 9.2.3). Several PALcode routines, including those for AMOVRR and AMOVRM, contain the RC instruction. Currently, the AMOVRR PALcode routine is not used.

Bracketing an instruction sequence with the RS instruction and either CALL_PAL AMOVRM or the RC instruction makes it possible to determine whether a sequence of instructions executed without incurring an exception or being interrupted.

Executing the RS instruction sets a per-processor state bit called intr_flag and returns its previous state in the instruction's register operand. The intr_flag state bit is cleared by the PALcode REI routine. Executing the RC instruction clears the intr_flag state bit and returns its previous state in the instruction's register operand. If the bit was already clear, an interrupt or exception occurred during the execution of the bracketed code sequence. The intr_flag state bit is also tested and cleared during the execution of the atomic move PALcode routines.

The atomic move PALcode routines enable multiple memory stores to be made atomically: all stores happen, or none does. Each routine supports a set of stores to multiple memory locations that is atomic with respect to threads of execution running on the same CPU.

The instruction CALL_PAL AMOVRR invokes the AMOVRR PALcode routine, whose function is to store the contents of its two source registers (R16 and R19) at two arbitrary memory addresses. The data stored from each register can be its low byte, word, longword, or the entire quadword. No alignment constraints are placed on the two addresses. R17 and R20 specify the destination addresses. The low-order two bits of R18 and R21 specify whether a byte, word, longword, or quadword of each source register should be stored.

The instruction CALL_PAL AMOVRM invokes the AMOVRM PALcode routine. It stores the low byte, word, longword, or the entire quadword of its one source register, R16, at the address specified by R17. R18 specifies the amount of data stored. In addition, the routine copies 0 to 63 aligned longwords from the memory location specified by R19 to the location specified by R20. R21 specifies the number of longwords to be copied. Either the store and the copy are both done, or neither is done. R18 is set to 1 if both are done, and to 0 otherwise.

Each of the atomic move PALcode routines first tests the intr_flag state bit. If the intr_state flag is clear, the routine clears R18 and exits, without doing any stores. If the intr_state flag is set, the routine clears it and then checks whether any memory access would cause a page fault, fault-on-write, fault-on-read, or access violation. If so, it performs no store and generates the corresponding fault. If the memory accesses can be completed without faulting, the routine attempts the stores using load-locked/store-conditional sequences. If all sequences store successfully with no interruption, the routine completes with R18 set to 1. If not, the routine completes with R18 set to 0.

A typical code sequence using either of these routines begins with an RS instruction, followed by initialization of the source and destination address registers, a CALL_PAL AMOVRM instruction, and a branch to reexecute the sequence if the stores were not successful.

The TIE$SHARE code that interprets untranslated VAX instructions maintains a VAX state save area. After interpreting each VAX instruction, the interpreter executes the CALL_PAL AMOVRM instruction to write the instruction result and update the VAX state save area atomically.

### 9.2.3 VAX Instruction Atomicity

As described in Chapter *Synchronization Techniques*, the VAX architecture guarantees that, once started, certain instructions execute without interruption. If such an instruction incurs a fault, any side effects are rolled back so that the instruction can be restarted from the beginning after the fault is handled.

Most VAX instructions are translated into a sequence of Alpha instructions. Although each Alpha instruction is noninterruptible and restart-

able after a fault, a sequence of instructions is not. The VEST utility and TIE$SHARE cooperate to emulate this aspect of VAX instruction execution.

A VAX program that requires instruction atomicity must be translated with the VEST qualifier /PRESERVE=INSTRUCTION_ATOMICITY. In response to this qualifier, the VEST utility translates code as though the qualifier /OPTIMIZE=NOSCHEDULE were present and does not rearrange generated code to improve performance. Moreover, where necessary, it inserts an RS instruction at the beginning of a sequence of Alpha instructions that modifies a VAX memory operand and performs the memory update with the AMOVRM PALcode routine. If an interrupt or exception occurs during the sequence, the memory modification is not made and the sequence is restarted.

In response to the qualifier /PRESERVE=INSTRUCTION_ATOMICITY, the VEST utility also creates an instruction granularity bitmap. Each bit in the instruction granularity bitmap represents one Alpha instruction in the translated part of the image. Bit 0 represents the instruction at offset 0 in the translated part of the image; bit 1 represents the instruction at offset 4; and so on. A set bit means that the corresponding Alpha instruction is the first instruction in a set of instructions that emulates a given VAX instruction. Each image translated with that qualifier has its own bitmap. Given a PC within the translated part of the image, TIE$SHARE can determine the boundaries of the set of instructions that contains it. Section 9.6.7 describes the use of the instruction granularity bitmap in dealing with delivery of a translated AST during execution of a set of Alpha instructions emulating a single VAX instruction.

## 9.3 TRANSLATING AN IMAGE

The manual *DECmigrate for OpenVMS AXP Systems Translating Images* describes the use of the VEST utility in detail. The sections that follow summarize VEST's actions and describe its output.

### 9.3.1 Translation Process

The VEST utility processes an OpenVMS VAX image in several main phases:

1. It analyzes the image file to find entry points and distinguish code from data. It determines entry points from global and debug symbol tables in the image and the image's transfer address as well as from optional input files created by the user or TIE$SHARE during a previous execution of the image. Beginning at each entry point it disassembles the image, recursively tracing the program flow. It breaks the flow into sequences of instructions called basic blocks and records information such as condition code use, register use, and stack depth within each block.

2. If VEST encounters a CALL, JSB, or JMP destination that is computed at run time or that is unknown, it generates a call to TIE$SHARE to identify the destination at run time (see Section 9.6.1).

   To deal with destinations in shareable images, VEST parses the VAX image's fixup vector table and transforms VAX-style transfers through fixup vector G^ and .ADDRESS entries to transfers using linkage pairs. It creates the associated fixup records so that linkage pair contents will be replaced with symbol vector entry contents from shareable images during image activation.

   Resolving destinations in shareable images at run time allows a shareable image to be replaced with an upwardly compatible one and also enables the determination of whether the shareable image is translated or native to be postponed until run time.

   Destinations that correspond neither to found entry points nor to shareable image entry points can represent system service requests or transfers of control to VAX code that may require interpretation at run time.
3. VEST generates Alpha code equivalent to the VAX code, optimizing wherever possible. For example, although executing each VAX instruction sets condition codes that can be tested, VEST does not generate an emulation of a condition code that is not used by the code being translated.
4. VEST generates calls to procedures in TIE$SHARE to control image execution and to emulate complex VAX instructions.

VEST builds an image containing both the original code and the generated code. Such an image is referred to as a translated image.

### 9.3.2 The Translated Image

A translated image contains the original VAX code and data, and the translated code, as well as data structures built by VEST for use during execution. A translated image undergoes image activation just as a native image does. VEST encapsulates the image body from a VAX image in a file that resembles a native image built by the OpenVMS Alpha linker. The encapsulated image body includes the fixup section but not the debug and other symbol tables. Chapter 8 discusses in detail the organization of a native image and structures within it.

The virtual address grain of a translated image is always 64 KB; that is, each image section begins on a $10000_{16}$ byte boundary. Although all current Alpha CPUs support a page size of 8 KB, a translated image with a grain of 64 KB could execute on any future Alpha CPU with a page size larger than 8 KB.

VEST merges image sections from the VAX image to the extent possible. For example, if a read-only image section and a writable image section are within the same grain, VEST makes the corresponding image section

in the translated image writable. It cannot merge VAX image sections with incompatible attributes such as copy-on-reference and shareable. Successful translation may require a VAX image to be relinked with the linker qualifier /BPAGE=16.

VEST generates one or more image sections that contain all the image sections from the VAX image, one image section that contains the translated code, and one that contains linkage and data for use by the translated code and TIE$SHARE during image execution. Data cells and areas statically declared within the VAX image retain their original addresses. The original VAX instructions may be necessary for run-time interpretation.

A translated image can be distinguished from a native image by a bit in the image header (EIHD$V_XLATED in EIHD$L_LNKFLAGS). Also, the default file name of a translated image has the string _TV appended to it, for example, MONITOR_TV.EXE. (TV stands for Translated from VAX.) Another typical difference is that a translated image begins at virtual address 0, whereas native images by default begin at 64 KB, or $10000_{16}$, rather than at virtual page 0. (Recall that the page size on current Alpha processors is 8 KB. Leaving virtual page 0 as a no-access page would make it impossible to map a translated image at the same address range on an Alpha processor as on a VAX processor.)

Figure 9.1 shows the structure of a translated executable image file, and Figure 9.2, that of a translated shareable image file. (Not all parts of these images are mapped into memory when the images are activated.) Every translated image includes TIE$SHARE in its shareable image list (see Chapter 8).

A translated shareable image includes a symbol vector and a global symbol table built by the VEST utility. The first entry in the symbol table is always reserved for nonstandard interimage references, for example, through a VAX JSB instruction. After relocation the entry contains the base address of the

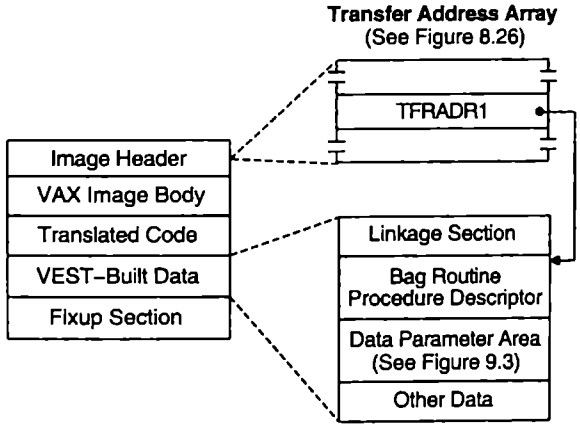



**Figure 9.1**
Organization of a Translated Executable Image

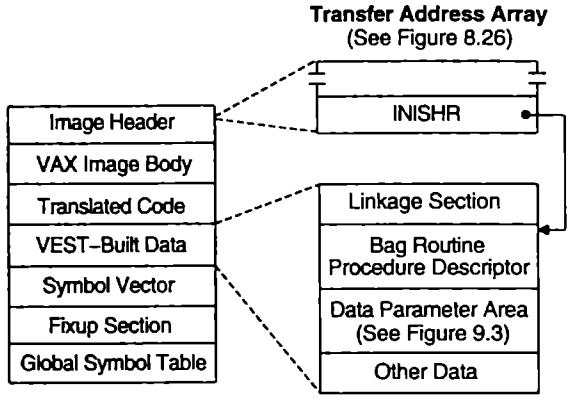



**Figure 9.2**
Organization of a Translated Shareable Image

shareable image. When a translated shareable image is replaced by a native shareable image, the first symbol vector entry in the native shareable image must be reserved for upward compatibility. It can be reserved through the SPARE keyword on the SYMBOL_VECTOR clause in the linker options.

A translated image also includes a native routine called the bag that VEST defines to be the first code executed within the image (see Section 9.5).

Data structures built by VEST include the following:

- A VAX-to-Alpha instruction map for lookups of computed destinations, for example, `JSB (R5)`, and of other destinations where VEST did not find the original VAX code
- Optionally, an instruction granularity bitmap, described in Section 9.2.3
- A data parameter area that describes the image

Figure 9.3 shows the layout of the data parameter area. Its fields are defined in [TIE]TIE.R64 and begin with the string TIE$.

TIE$A_IMAGE_BASE contains the base virtual address of the translated image. TIE$L_ALPHA_START contains the offset in the image of the start of the translated code within the image, and TIE$L_ALPHA_END contains the image offset of its end.

TIE$L_VIA_MAP contains the image offset of the VAX-to-Alpha instruction map. TIE$L_BOUNDARY_MAP contains the offset to the instruction granularity bitmap, if present. Otherwise, it contains the value $80000000_{16}$.

For an executable image, TIE$A_ROUTINE contains the image offset of the entry point in the translated code.

The fields that contain image offsets are relocated at image activation and thus contain virtual addresses at run time.

TIE$T_VEST_ID is a counted ASCII string identifying the version of the VEST utility that translated the image. TIE$T_VEST_DATE_TIME is

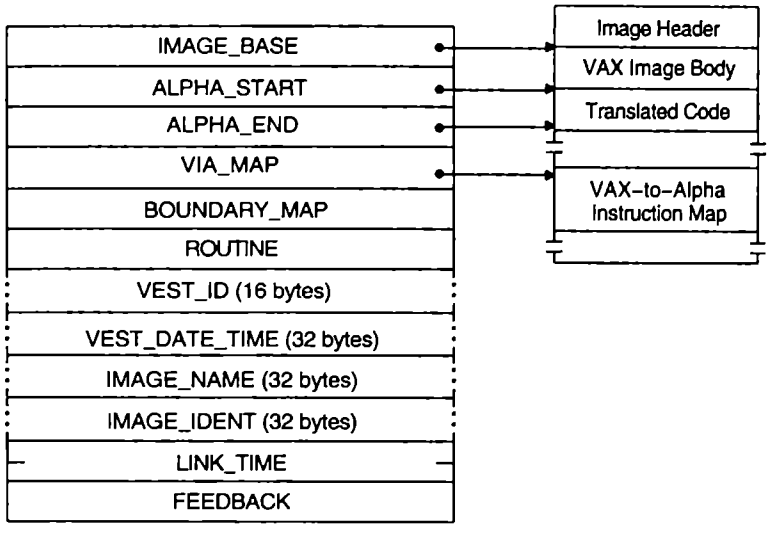



**Figure 9.3**
Layout of the Translated Image Data Parameter Area

a counted ASCII string of the date and time at which the image was translated.

TIE$T_IMAGE_NAME is a counted ASCII string containing the name of the original VAX image. TIE$T_IMAGE_IDENT is a counted ASCII string identifying the version of the image. TIE$Q_LINK_TIME contains the time in system time format when the image was linked.

The low bit of TIE$L_FEEDBACK, when clear, indicates the image was translated with the /FEEDBACK qualifier.

### 9.3.3 Constraints on Candidate Images

Designed for user mode images, the VEST utility does not translate an image that contains user-written system services, or references to system space or P1 space addresses other than OpenVMS VAX system service vector addresses. VEST cannot translate based shareable images or any image that makes an absolute reference to a P0 space address not within it.

VEST will not translate images linked on VAX/VMS versions prior to Version 4.0.

VEST does translate an image that contains VAX instructions requiring kernel mode, for example, MTPR; the bugcheck instructions (BUGW and BUGL); and VAX vector instructions. A run-time error will result, however, if an attempt is made to execute any of them.

It cannot translate images with source modules written in VAX Ada because VAX Ada programs perform certain stack manipulations as part of thread multitasking that are unsupported by the translated image run-time environment.

## 9.4 CALLING STANDARDS AND JACKETING

The OpenVMS VAX and Alpha calling standards differ significantly. While transparent to a high-level language programmer, these differences make it difficult simply to translate between VAX calls and Alpha calls. Differences include

- Form of procedure frames
- Definition of a procedure value
- Method of passing arguments

A VAX procedure or call frame is built on the stack and may include the procedure's arguments, depending on the form of the call. When a procedure is entered, the FP register points to its call frame, built by a CALLG or CALLS instruction. Optionally, the call frame can specify the address of a condition handler specific to that procedure invocation. A routine entered with a JSB instruction has no stack frame and, indeed, is not a procedure.

In contrast, there are three types of Alpha procedure, distinguished by the existence and form of their procedure frames: stack frame, register frame, and null frame. Compiler-generated code at the beginning of the first two types of procedure builds a procedure frame. (A null frame procedure implements JSB-type transfers and has no frame.)

The procedure value of a VAX procedure is the address of its entry mask, which specifies registers to be saved. The procedure value of an Alpha procedure is the address of its procedure descriptor, a data structure that identifies the procedure type, describes its register and stack use, and identifies its code entry point.

Arguments to a VAX procedure are passed in a memory list built by the caller. The list may be on the stack as part of the call frame. The call stores the address of the argument list in the argument pointer (AP) register, which the procedure uses as a pointer to the list.

In contrast, the first six arguments to an Alpha procedure are passed in registers, and any others are passed in stack locations. The argument information (AI) register specifies the number of arguments and identifies the types of the first six. Integers in the first six arguments are passed in integer registers, and floating-point numbers in floating-point registers.

Chapter *System Overview* introduces procedure types and other OpenVMS Alpha calling standard conventions. The *OpenVMS Calling Standard* manual provides a more complete description.

Despite the calling standard differences, translated and native images that follow their respective calling standards can interoperate transparently. Code in a translated image can call native procedures as well as translated procedures; code in a native image can call native and translated procedures. A shareable translated image can be replaced with a native shareable image without the need to relink or retranslate any of its callers (assuming the order of entries in the native symbol vector matches that of the translated symbol

vector). Similarly, a shareable native image can be replaced with a translated shareable image.

Translated and native images that do not follow their respective calling standards may still be able to interoperate. Jacketing images may need to be interposed between them.

The sections that follow briefly describe support for interoperability.

### 9.4.1 Relevant OpenVMS Alpha Calling Standard Features

The OpenVMS Alpha calling standard provides the following features to support interoperability:

- A way to determine whether a given procedure value represents a native or a translated procedure
- A way to determine whether a given procedure frame encapsulates translated frames
- A mechanism to describe all the arguments input to a procedure and the form and location of the value returned by the procedure
- A mechanism to enable calling a procedure through an intermediary or bridge procedure

The procedure value of a native procedure is the address of its procedure descriptor, which begins with a word of flags. The calling standard specifies that bit 12 (PDSC$V_NATIVE) of the flags must be set to 1 for native compiled code. The procedure value of a translated procedure is the address of its entry mask. The VAX architecture specifies that bit 12 in that mask must be 0. Thus, testing bit 12 at the address specified by the procedure value distinguishes the two types of procedures: when set, the procedure is native; when clear, it is translated.

Another procedure descriptor flag bit (PDSC$V_TIE_FRAME) is set to indicate a TIE$SHARE procedure frame that encapsulates one or more translated frames (see Section 9.6.2).

The OpenVMS Alpha calling standard defines an optional extension to the procedure descriptor called procedure signature information. The additional description specifies the number, type, and location of arguments, and the form and location of the value returned by the procedure. A procedure descriptor includes a field for the offset to its procedure signature information, if any. The value 0 in this field indicates no signature. The value 1 indicates the procedure has a default signature. The signature information enables software to transform a VAX-style argument list into the arguments expected by a native procedure, and any returned function value into the form expected by the translated caller, or to translate native arguments into VAX argument format.

A default signature for a native procedure called by a translated procedure has the following meaning:

- The count in the VAX argument list is the number of input arguments. The count is copied to the AI register.
- All arguments, if any, are 32-bit sign-extended integers.
- The function result, if any, is a 32-bit sign-extended integer.

The OpenVMS Alpha calling standard provides bound procedures and bound procedure descriptors, partly to enable a target procedure to be called through an intermediary or bridge procedure. A bound procedure descriptor has a different format from a normal procedure descriptor. The bridge procedure it describes must be simple with no frame or arguments of its own. A bound procedure descriptor includes a field to point to the descriptor of the target procedure. The address of a bound procedure is bound at link time, but the address of the target procedure descriptor is not bound until image activation; the binding may even be dynamically calculated at run time. Bound procedure descriptors interpose jacketing procedures between native and translated procedures.

The bridge procedure code is entered like any other procedure with the address of its procedure descriptor in R27. Like any other procedure descriptor, a bound procedure descriptor can contain the offset of a procedure signature. This makes it possible to connect procedure signature information from the calling procedure with the target procedure. The bridge procedure makes it possible to transform the format of the caller's arguments to the format expected by the target procedure.

A default signature associated with a call from a native procedure through a bound procedure descriptor to a translated procedure has the following meaning:

- The AI register contains the number of input arguments, which is copied to the VAX argument list count.
- The register arguments, if any, are described by the AI register.
- All memory arguments, if any, are 32-bit sign-extended integers.
- The function result, if any, is a 32-bit sign-extended integer.

### 9.4.2 Automatic Jacketing

A jacket procedure is interposed between caller and target procedures when one is native and the other translated. The jacket procedure transforms the input arguments and, if necessary, the returned function value from one architecture to another. How and when the jacket is interposed depends on the nature of the image containing the caller procedure.

The modules of a native image that call translated procedures or that are called from translated code must be compiled with the /TIE qualifier. In response to that qualifier, the compiler generates procedure signature information for each native procedure. The procedure signature information will be used in any inbound call from a translated procedure to transform its VAX-style argument list to the form expected by the native procedure.

The compiler also compiles outbound calls differently. In particular, it creates a signature for each outbound call and generates an object language record associating the signature with the linkage pair describing the target procedure. When such a native image is linked with the /NONATIVE_ONLY qualifier, the linker creates a special fixup record for each outbound call. The fixup record is called a fixup for a linkage pair with a procedure signature block (see Chapter 8).

When such an image is activated along with one or more translated images, the image activator processes those fixup records and creates virtual address space for bound procedure descriptors and procedure signature blocks. Before the image and all associated shareable images are activated, it is not possible to determine whether the target of the outbound call is native or translated. After all images have been activated, the fixup records are processed. For each fixup record whose target procedure is in a native image, a standard linkage pair fixup is done so that the native image caller will call the target native procedure directly.

For each fixup record whose target procedure is in a translated image, a bound procedure descriptor is built and, optionally, a procedure signature block. The bound procedure descriptor represents a request of the $NATIVE_TO_TRANSLATED system service. The procedure bound to it is the target translated procedure. The linkage pair is fixed up to point to the bound procedure descriptor. At run time, the native caller will request the system service in order to call the target translated procedure.

Figure 9.4 illustrates the connections among the procedures.

The mechanism just described serves to jacket each call that meets all the following criteria:

- Its target procedure is known at link time.
- The number of arguments is fixed.
- It does not require CALLG emulation.

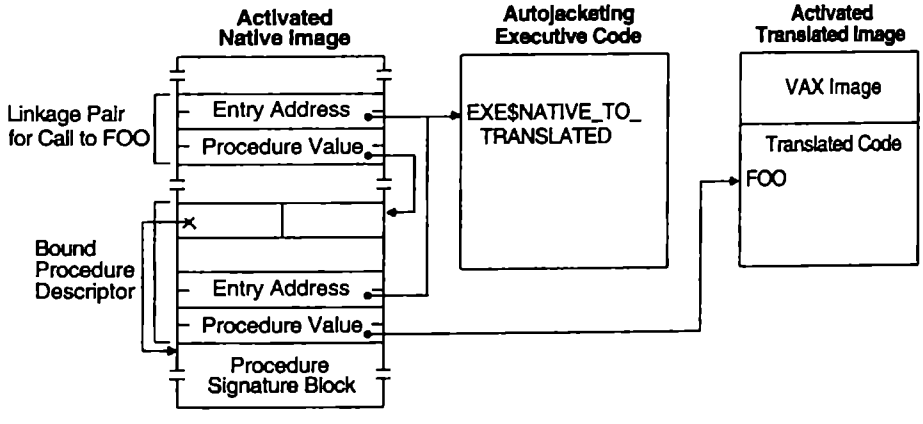



**Figure 9.4**
Jacketing from a Native Procedure to a Translated Procedure

For calls that do not meet those restrictions, compilers generate calls to intermediate procedures.

Whether or not a module is compiled with the /TIE qualifier, a compiler generates a call to OTS$CALL_PROC, passing it the target procedure value, whenever a call meets all the following criteria:

- Its target procedure destination is not known at compile and link time; for example, it is a computed destination.
- The number of arguments is fixed.
- It does not require CALLG emulation.

If such a module is compiled /TIE, the compiler generates procedure signature information associated with the linkage pair for OTS$CALL_PROC. If the image is then linked /NONATIVE_ONLY, the linker generates a linkage pair fixup with procedure signature information. Because OTS$CALL_PROC is in a native image, at image activation a standard linkage pair fixup is done. Therefore, at run time the caller calls OTS$CALL_PROC directly.

OTS$CALL_PROC, in module [LIBOTS]OTS$CALL_PROC_ALPHA, checks at run time whether the target procedure is native or translated. If native, it calls the procedure. If translated, it requests the $NATIVE_TO_TRANSLATED system service to call the procedure.

For a call that has a variable number of arguments or that requires CALLG emulation, a different intermediate procedure is used. The procedure is language-specific. The MACRO-32 compiler generates a call to AMAC$EMUL_CALL; the BLISS-32 compiler generates a call to BLI$CALLG; other compilers use equivalent language-specific procedures. Whatever the procedure, it is passed the target procedure value. If the module containing the call is compiled /TIE, the compiler also generates procedure signature information associated with the linkage pair for AMAC$EMUL_CALL or its equivalent. If the image is linked /NONATIVE_ONLY, the linker generates a linkage pair fixup with procedure signature information. Because AMAC$EMUL_CALL and its equivalents are in native images, at image activation a standard linkage pair fixup is done. At run time the caller calls AMAC$EMUL_CALL or its equivalent directly, whether or not the target procedure is translated.

The module containing AMAC$EMUL_CALL itself is compiled /TIE and contains a computed call to the target procedure whose address is passed. The call is therefore compiled as a call to OTS$CALL_PROC. When AMAC$EMUL_CALL is called, it forms the argument list, performs whatever call emulation is required, and then calls OTS$CALL_PROC. OTS$CALL_PROC tests whether the target procedure is native or translated and, if translated, requests the $NATIVE_TO_TRANSLATED service to call the procedure. Similar considerations apply to the other language-specific procedures equivalent to AMAC$EMUL_CALL.

Section 9.4.3 describes how the executive calls translated procedures.

All external calls made by a translated image are processed by TIE$SHARE. It checks whether the target procedure is translated or native. If native, it requests the $TRANSLATED_TO_NATIVE system service to call the native procedure (see Section 9.4.5).

### 9.4.3 Jacketing Initiated by the Executive

Several executive routines call procedures specified by user images. These routines check whether the procedure is native or translated and, if translated, call the procedure through the $NATIVE_TO_TRANSLATED system service.

When delivering an AST to an outer mode (that is, not kernel), the AST delivery interrupt service routine tests bit 12 at the procedure value address. If the bit is set, the AST procedure is a native one and is called in the usual way (see Chapter *ASTs*). If the bit is clear, the AST procedure is a translated one. The interrupt service routine executes a `CALL_PAL REI` instruction to return to the outer mode, specifying the code entry point of the $NATIVE_TO_TRANSLATED service as the return address. The interrupt service routine passes the target AST procedure value and the value 1 to indicate a default procedure signature block. The $NATIVE_TO_TRANSLATED service calls the translated AST procedure.

The AST interrupt service routine is unusual in that it is coded in MACRO-64 assembly language. Other executive and RTL routines that call user-specified procedures are written in MACRO-32 or higher level languages and are compiled with the /TIE qualifier. As a result, their calls to user-specified procedures are done through OTS$CALL_PROC or procedures such as AMAC$EMUL_CALL and BLI$CALLG.

### 9.4.4 Jacketing Prohibited by the Executive

To help enforce the constraint that translated code must not execute in inner modes, the $CMEXEC and $CMKRNL system services specifically check that the descriptor of the target procedure has PDSC$V_NATIVE set and PDSC$V_TIE_FRAME clear. If either test fails, the service returns the error status SS$_BADPARAM. These tests ensure that the procedure to be entered in inner mode is native and not a procedure that encapsulates a translated procedure.

The Set Exception Vector ($SETEXV) system service confirms that the procedure being established as a primary, secondary, or last chance handler is a native procedure. If not, the service returns the error status SS$_IVSSRQ.

### 9.4.5 Automatic Jacket System Services

The jacket system services are mode of caller services in module SYSJACKET.

The $NATIVE_TO_TRANSLATED service is requested with the arguments for the translated procedure in the standard native form (the argument count in AI, the first six arguments in R16 through R21, and arguments beyond six on the stack). In addition, at entry to the service, R23 contains the address of the translated entry point and R24 identifies the signature associated with this call.

EXE$NATIVE_TO_TRANSLATED, the system service procedure for the $NATIVE_TO_TRANSLATED service, takes the following steps:

1. It executes a TRAPB instruction so that any pending exceptions will be delivered in the procedure context of the service's requestor.
2. It builds its own stack frame and establishes itself as the current procedure.
3. It determines the current access mode. If the current mode is executive, it generates the nonfatal bugcheck NOCALLTRANS and requests the $EXIT system service, which will delete the process. If the current mode is kernel, it generates the fatal bugcheck NOCALLTRANS. These tests help enforce the constraint that translated code must not execute in inner modes.

   If the current mode is supervisor, it generates the exception SS$_NOCALLTRANS_SUP.

   If the current mode is user, it continues.
4. It checks that either a default signature was specified or a procedure signature block is present, signaling the error SS$_NOCALLTRANS if not. This error typically occurs under two circumstances:
   - The source module containing the service's requestor was not compiled with the /TIE qualifier.
   - The image containing that module was linked with the qualifier /NATIVE_ONLY rather than /NONATIVE_ONLY.
5. It analyzes the procedure signature, if one is present, determining the number and types of arguments. It allocates stack space for a VAX-style argument list with one longword for each argument and one for the argument count. If more than six arguments were passed from the native caller, the additional arguments are on the stack, at addresses older than the stack frame built in step 2. In the case of a target procedure that returns a double-complex function value, it reserves 16 bytes for the returned data and passes the address of the 16 bytes as the first argument.

   EXE$NATIVE_TO_TRANSLATED copies the low-order longword of each additional argument to the allocated stack space. It copies the arguments passed in registers to the allocated stack space, for example, storing the low-order longword of an integer argument. It stores the argument count in the first longword of the list.
6. It aligns the stack to an octaword boundary preparatory to a call, loads the address of the argument list into R16, loads the address of the

translated entry point into R17, and calls the procedure whose value is in CTL$GL_NATIVE_TO_TIE, namely, TIE$NATIVE_TO_TIE, a TIE$SHARE procedure in module [TIE]TIELOOKU.

TIE$NATIVE_TO_TIE builds a procedure frame to encapsulate the VAX procedure frame, saves its caller's registers, establishes a VAX environment for the translated procedure, and transfers control to it (see Section 9.6). The translated procedure returns to TIE$NATIVE_TO_TIE, which restores saved registers and returns.

7. When TIE$NATIVE_TO_TIE returns, if a nondefault procedure signature block was present, EXE$NATIVE_TO_TRANSLATED checks whether a function value is being returned and, if so, takes the necessary steps to load it from the stack into R0 or F0.
8. EXE$NATIVE_TO_TRANSLATED cleans up the stack, restoring saved registers from the stack frame, and returns to its requestor. If more than six arguments were passed, the requestor is responsible for removing those on the stack.

The $TRANSLATED_TO_NATIVE system service is only requested by the TIE$SHARE procedure TIE$$OUTBOUND_JACKET, in module [TIE]TIEEXC. The service is requested with two arguments: the address of a VAX-style argument list and the procedure value of the target procedure.

EXE$TRANSLATED_TO_NATIVE, the system service procedure for the $TRANSLATED_TO_NATIVE service, takes the following steps:

1. It executes a TRAPB instruction so that any pending exceptions will be delivered in the procedure context of the service's requestor.
2. It builds its own stack frame and establishes itself as the current procedure.
3. It checks that the target procedure has a default signature or an associated procedure signature block, signaling the error SS$_TRANSCALLER if not. This error typically occurs when the source module containing the target procedure was not compiled with the /TIE qualifier.
4. If the signature is a default one, it loads as many integer argument registers as there are arguments and copies the argument count to the AI register, zeroing the rest of it to indicate integer arguments.

   If the VAX-style argument list contains more than six arguments, it allocates the stack space for them, ensuring the stack remains octaword-aligned. It copies the seventh and succeeding arguments to the stack, sign-extending each longword argument to a quadword.
5. If the signature is a nondefault one, EXE$TRANSLATED_TO_NATIVE analyzes the procedure signature, determining the type and size of each argument. If the target procedure returns a double-complex function value, EXE$TRANSLATED_TO_NATIVE saves the first argument, which is the address of 16 bytes in which the data is to be returned, and decrements the argument count. In the case of an integer quadword input

argument or a D or G floating-point argument, it forms one quadword argument from two longwords in the argument list.

It loads the first six arguments into integer or floating-point argument registers as appropriate and stores the others as quadwords in allocated stack space.

If the caller passed more arguments than specified in the procedure signature block, EXE$TRANSLATED_TO_NATIVE treats them as longword integers.

It initializes the AI register with the number of arguments actually present and the type (floating or integer) of the first six.

6. It calls the specified native procedure.
7. When the native procedure returns, EXE$TRANSLATED_TO_NATIVE cleans up the stack, restoring saved registers from the stack frame.
8. If the target procedure has a nondefault signature, EXE$TRANSLATED_TO_NATIVE performs any necessary processing on the function value returned, if present.
9. It returns to its requestor. (Actually, because of the manner in which TIE$$OUTBOUND_JACKET calls the service, it returns to the caller of TIE$$OUTBOUND_JACKET.)

## 9.5 TRANSLATED IMAGE INITIATION

TIE$SHARE specifies the routine TIE$SHARE_INIT, in module [TIE]TIE, as its initialization procedure. Consequently, whenever a set of activated images includes one or more translated images and thus TIE$SHARE, TIE$SHARE_INIT executes. Its major function is to record the procedure value of TIE$NATIVE_TO_TIE in the cell CTL$GL_NATIVE_TO_TIE for subsequent use by the $NATIVE_TO_TRANSLATED system service.

### 9.5.1 Translated Executable Image Initiation

When a translated executable image is activated and then invoked, several routines execute before any translated code:

1. The bag routine specified by the transfer address in the image's header
2. TIE$SETUP_IMAGE, in module [TIE]TIEIMG
3. TIE$$SETUP_IMAGE, in module [TIE]TIE
4. The jacketing system service $NATIVE_TO_TRANSLATED
5. TIE$NATIVE_TO_TIE

Like most procedures, the bag routine is entered with its procedure value in R27. It is a null frame procedure and is called by the Digital command language (DCL) interpreter responding to a RUN command or by EXE$PROCSTRT (see Chapter 7). In either case, it is called with the arguments described in Chapter 11.

The data parameter area, shown in Figure 9.3, immediately follows the bag routine's procedure descriptor. The bag routine consists of several instructions that make a nonstandard call to TIE$SETUP_IMAGE, part of TIE$SHARE, passing it the arguments with which the bag routine was entered. The nonstandard call preserves the contents of R27 for TIE$SETUP_IMAGE's use as a pointer to the parameter area.

TIE$SETUP_IMAGE builds a stack frame, saving in it the bag routine's procedure value and arguments. Dealing with the nonstandard call and building the stack frame in a particular way requires MACRO-64 code. TIE$SETUP_IMAGE calls TIE$$SETUP_IMAGE to do the image-specific initialization that does not require MACRO-64 code.

TIE$$SETUP_IMAGE creates the first entry in an in-memory table to describe the image. It requests the Set Fault ($SETFLT) system service to make every page in the VAX portion of the image a fault-on-execute page. Any subsequent attempt to execute code in such a page will cause an exception that will be dispatched to a condition handler in TIE$SHARE (see Section 9.6.6). TIE$$SETUP_IMAGE returns to TIE$SETUP_IMAGE.

Figure 9.5 shows the layout of the entry built by TIE$$SETUP_IMAGE to describe the image.

Its fields are defined in [TIE]TIE.R64 and have no facility prefix. The entry consists largely of data copied from the image's data parameter area, described in Section 9.3.2, and pointers to fields in the data parameter area. For example, VEST_DATE_TIME contains the address of the VEST date and time stamp in the parameter area, and VEST_ID, the address of the ID of the VEST image used to translate this image.

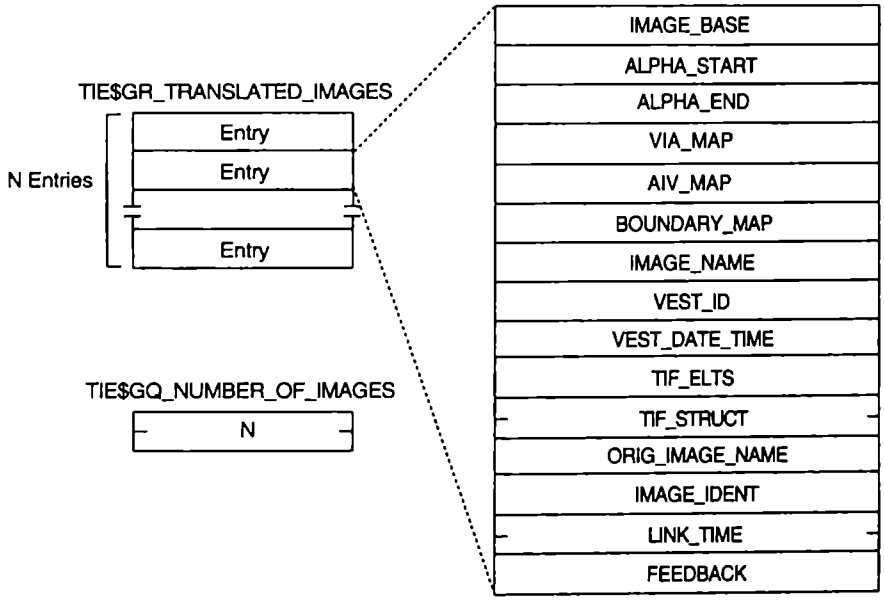



**Figure 9.5**
Layout of a Translated Image Descriptor Entry

ORIG_IMAGE_NAME contains the address of the original VAX image name in the parameter area.

AIV_MAP contains the address of the Alpha-to-VAX instruction map, an inverted form of the VAX-to-Alpha instruction map.

IMAGE_NAME contains the address of this image's name in the image control block that describes it (see Chapter 8).

TIF_ELTS contains the maximum number of elements in the tree. TIF_STRUCT is the listhead for entries that contain feedback information generated during translated image execution. Each entry identifies untranslated VAX code found at execution and the type of transfer made to the code, for example, call or jump. At image rundown, the feedback information is written to a file that can be input to a subsequent translation for improved performance.

TIE$SETUP_IMAGE examines TIE$A_ROUTINE in the translated image data area to get the address of the routine to call in the translated image. The address corresponds to the routine specified by the image transfer address in the original VAX image. It restores the argument registers and copies the transfer address array (see Figure 8.26) to its own stack frame so that the translated image's procedure can reference them as an array of longwords. It requests the $NATIVE_TO_TRANSLATED service, passing it the addresses of a default procedure signature block and the translated image's entry point.

As described in Section 9.4.5, the $NATIVE_TO_TRANSLATED system service calls TIE$NATIVE_TO_TIE with the address of the translated routine's entry point. TIE$NATIVE_TO_TIE, described in Section 9.6.2, transfers control to the translated code.

### 9.5.2 Translated Shareable Image Initiation

A shareable translated image goes through the same first three steps as an executable translated image. During image activation, after an executable image and all related shareable images have been mapped and address relocations and fixups done, shareable image initialization procedures are called. A translated shareable image specifies the bag routine as its initialization procedure. The bag routine calls TIE$SETUP_IMAGE, which calls TIE$$SETUP_IMAGE. TIE$$SETUP_IMAGE adds an entry for the shareable image to the in-memory table that describes activated translated images.

A shareable image has no transfer routine, and control returns from these procedures to image activator code.

## 9.6 EXECUTION WITHIN THE TRANSLATED ENVIRONMENT

A translated executable image has one entry point, namely, its transfer routine. During its execution it can declare procedures within itself as callable in response to events such as ASTs and exceptions.

A translated shareable image has multiple entry points defined within a symbol vector and global symbol table. It can also declare procedures within itself as callable in response to ASTs and exceptions.

Either type of translated image includes executable code sequences and transfers of various kinds. These transfers include the following:

- Transfers from native to translated images, including the initial entry
- Transfers within and among translated images
- Transfers from translated to native images
- Transfers from a translated image to OpenVMS system services

Native and translated codes share the same stack, with VAX call frames and native procedure frames interleaved by TIE$SHARE.

The sections that follow describe how transfer destinations are located and the various kinds of transfers.

### 9.6.1 Locating Transfer Destinations

VEST generates a call to one of several TIE$SHARE lookup routines for any transfer of control from a translated image to an unknown destination. These include TIE$JMP_LOOKUP, TIE$JSB_LOOKUP, and TIE$CALLX_LOOKUP, all in module [TIE]TIELOOKU. TIE$NATIVE_TO_TIE calls TIE$CALLX_LOOKUP in transferring from native to translated code.

All the lookup routines can be used for transfers among routines in translated images. TIE$CALLX_LOOKUP, however, is the only routine that can transfer from one domain to another, that is, from translated code to native code, or vice versa. In other words, the only supported way to transfer control between domains is through a procedure call.

Each lookup routine first examines a 1,024-entry lookup cache of frequently referenced destinations. If the destination is not in the cache, each routine calls TIE$XXXX_LOOKUP, in module [TIE]TIE, to perform the actual lookup of the destination.

TIE$XXXX_LOOKUP first uses the translated image table (see Figure 9.5) to determine in which image, if any, the destination lies. The possible outcomes include the following:

- The destination is within a translated image but not within the translated part of that image. It is the address of a VAX instruction, possibly generated by some form of computed transfer in the image. In any case, it represents an entry point not found during translation.
- The destination is within the translated part of a translated image, and TIE$XXXX_LOOKUP looks up the address in the VAX-to-Alpha instruction map for that image. The instruction map table entry contains the corresponding translated code address.
- The destination is within TIE$SHARE.

- The destination is an address within the range of OpenVMS VAX system service vectors, and TIE$XXXX_LOOKUP transforms the address as described in Section 9.6.5.
- The destination is within a native shareable image.

After locating the destination, TIE$XXXX_LOOKUP updates the cache and returns to its caller with the translated address, if appropriate, and a status specifying whether the destination is translated, VAX code, or outside any translated images.

### 9.6.2 Transfers from Native to Translated Images

As previously described, when control is first transferred to an activated executable translated image, the $NATIVE_TO_TRANSLATED system service calls TIE$NATIVE_TO_TIE in order to enter the translated code. Most other transitions from native code to translated code follow the same sequence. The only other path for a transition from native to translated is described later in this section.

TIE$NATIVE_TO_TIE is written in MACRO-64 code so that it can control stack layout and register use. The descriptor for this procedure identifies it as a PDSC$V_NATIVE frame and as a PDSC$V_TIE_FRAME, one that encapsulates translated frames.

TIE$NATIVE_TO_TIE creates a special stack frame for saving registers and exception context. Definitions in module [TIE]TIE.PREFIX specify the layout of this area, shown in Figure 9.6.

Most of the fields in the frame are for saving registers. TIE_PDV contains the procedure value of the TIE$SHARE procedure that built the frame. TIE_

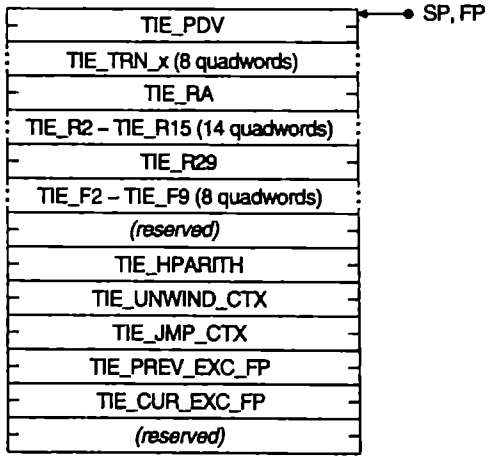



**Figure 9.6**
Layout of the TIE Frame Register Save Area

TRN_0 to TIE_TRN_7 are scratch locations for use by translated code. The other fields are used in condition handling.

TIE$NATIVE_TO_TIE is called with the addresses of the VAX argument list and the translated procedure to be called.

It takes the following steps:

1. It builds the register save area on the stack and records its address in the FP.
2. It calls TIE$$ALIGN_VAX_STATE, in module [TIE]TIESTATE, to initialize R12 and R13, the VAX AP and FP, from the register save area of the TIE frame containing the most recent VAX procedure. When translated code is first entered, there is no earlier VAX procedure, and R12 and R13 are simply cleared.
3. TIE$NATIVE_TO_TIE clears R22 and sets R23 to 1 to initialize the emulated VAX condition codes.
4. It copies the SP to R14, the VAX SP.
5. It calls TIE$CALLX_LOOKUP, in module [TIE]TIELOOKU, to transform the address of the VAX procedure to the address of its translated code and to enter the translated code or to transfer control to the interpreter if the lookup fails.

TIE$CALLX_LOOKUP calls TIE$XXXX_LOOKUP, described in Section 9.6.1. When TIE$XXXX_LOOKUP returns, TIE$CALLX_LOOKUP has two paths into a translated image, depending on what the destination address represents. In either case, it reserves space on the stack for a maximum-size VAX call frame and adjusts the SP to an octaword boundary. (This is somewhat simplified; if the Alpha SP is advanced enough beyond the VAX SP so that there is room for the VAX call frame, no adjustment is made to the Alpha SP.) If the address is within translated code, TIE$CALLX_LOOKUP transfers control to the translated code, which contains VEST-generated instructions to build a VAX call frame in the allocated space.

If the VAX procedure address is in VAX code, it presumably represents an entry point not found during translation. TIE$CALLX_LOOKUP evaluates the word at that address as a VAX procedure entry mask. It aligns R14, the VAX SP, to a longword boundary and builds a call frame, starting at R14 and saving the specified registers. It then looks up that address plus 2 as a branch destination in case the VAX code has been translated but not identified as a procedure entry point. If the lookup fails, it calls TIE$$INTERPRET_VAX, in module [TIE]TIELOOKU, which performs stack setup for interpretation and then calls the interpreter, TIE$$VAX_INTERPRETER, in module [TIE]VAX_INTERPRETER.

The only other transition from native to translated image code is through TIE$$N_TO_T_CH, in module [TIE]TIEEXC, which dispatches into a translated image condition handler following an exception. TIE$NATIVE_TO_

TIE's procedure descriptor identifies procedure TIE$$N_TO_T_CH as its frame-based condition handler. If an exception occurs while translated code is executing, the OpenVMS condition handling dispatcher dispatches to TIE$$N_TO_T_CH.

The descriptor for this procedure identifies it as a PDSC$V_NATIVE frame and as a PDSC$V_TIE_FRAME, one that encapsulates translated frames. TIE$$N_TO_T_CH creates the same special stack frame as TIE$NATIVE_TO_TIE (see Figure 9.6).

TIE$$N_TO_T_CH first checks whether the exception was caused by an attempt to access code in a fault-on-execute page (see Section 9.6.6). If not, it then must find the most recent VAX frame encapsulated within the TIE$NATIVE_TO_TIE frame that established TIE$$N_TO_T_CH as condition handler. If the condition is not handled by a translated code primary or secondary condition handler, TIE$$N_TO_T_CH must scan back through any encapsulated VAX call frames in that TIE$NATIVE_TO_TIE frame for a nonzero FP value.

### 9.6.3 Encapsulated VAX Call Frames

When one translated procedure calls another, VEST code generated by the caller builds a VAX call frame, and VEST code generated by the called procedure emulates the VAX RET instruction. As one translated procedure calls another and it calls another, nested VAX call frames build up on the stack. These VAX call frames are encapsulated within a native stack procedure frame; that is, from the viewpoint of the rest of the system, the current procedure context is that of TIE$NATIVE_TO_TIE. The VAX call frames are not a visible part of the procedure call chain.

Figure 9.7 shows an example stack with three VAX call frames encapsulated within TIE$NATIVE_TO_TIE's frame. The saved FP in the most recent VAX call frame, the one labeled VAX Call Frame 0, points to VAX Call Frame 1, and its saved FP points to VAX Call Frame 2.

### 9.6.4 Transfers from Translated to Native Images

As previously described, TIE$CALLX_LOOKUP is called by VEST-generated code to deal with any transfers of control whose destination is not within the same translated image. It calls TIE$XXXX_LOOKUP, described in Section 9.6.1, to locate the destination.

If the destination is within TIE$SHARE, TIE$CALLX_LOOKUP simply treats it as another translated procedure. It builds a VAX call frame and transfers control, as described in Section 9.6.2.

If the destination is an address within the range of OpenVMS VAX system service vectors, TIE$CALLX_LOOKUP takes the actions described in Section 9.6.5.

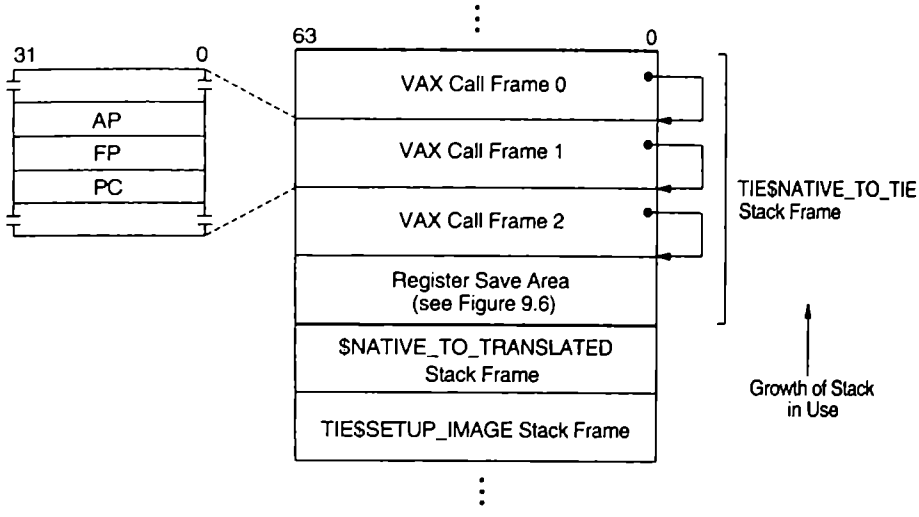



**Figure 9.7**
Translated Image Frames

If the destination is neither of these special cases, it examines bits 12 and 13 at that address. If either is set, the address is a native procedure value. TIE$CALLX_LOOKUP builds a VAX call frame, aligns the stack to an octaword boundary, and requests the $TRANSLATED_TO_NATIVE system service. When it returns, TIE$CALLX_LOOKUP emulates a VAX RET to return to its caller in translated code.

### 9.6.5 System Service Requests

In general, a system service request from a translated image takes the form of a call to an address that TIE$SHARE must transform. By default VAX system service requests are resolved to P1 space system service vectors. When the call destination is in this range, TIE$XXXX_LOOKUP indexes a table to transform the P1 space system service vector offset into the address of the equivalent OpenVMS Alpha system service. Actually, it transforms it into the procedure value of the equivalent system service transfer routine (see Chapter *System Service Dispatching*). TIE$SHARE then requests the $TRANSLATED_TO_NATIVE system service, specifying that procedure value and the address of the associated argument list.

That sequence works because the OpenVMS Alpha executive provides functionally equivalent system services and a system service interface that is upwardly compatible with the OpenVMS VAX interface. OpenVMS Alpha system services all have default procedure signatures. Nonetheless, several system service requests require explicit intervention:

- Change to Executive Mode ($CMEXEC)
- Change to Kernel Mode ($CMKRNL)

- Declare Change Mode or Compatibility Mode Handler ($DCLCMH)
- Set Exception Vector ($SETEXV)
- Unwind Call Stack ($UNWIND)

TIE$SHARE transforms requests for each of these to a procedure value of a procedure in TIE$SHARE whose name has the form TIE$*service-name*. It transfers control to any of these procedures through the $TRANSLATED_TO_NATIVE service.

TIE$CMEXEC and TIE$CMKRNL, in module [TIE]TIESYS, simply check whether the target inner mode procedure is an OpenVMS VAX system service vector (for example, SYS$ENQ). If so, the TIE procedure transforms it to the address of its counterpart OpenVMS Alpha transfer routine. Each then requests either the $CMEXEC or the $CMKRNL service.

Each of these system services explicitly checks if the target procedure is translated and if its descriptor has the bit PDSC$V_TIE_FRAME set. If either condition is true, it returns the error status SS$_BADPARAM. This means that a translated image can request these system services, but the target inner mode procedure must be a native procedure.

TIE$DCLCMH, in module [TIE]TIESYS, intercepts $DCLCMH requests. First, it makes two sanity checks on the request, returning the error status SS$_IVSSRQ if either fails:

- That the request is not intended to establish a compatibility mode handler
- That the handler being declared is a translated procedure

It then requests the $DCLCMH service, specifying its own procedure as the CHMU or CHMS handler. If the specified change mode exception occurs subsequently, its own handler will be entered and will transform the exception information and environment into what an OpenVMS VAX handler would expect, namely, a stack containing the sign-extended change mode operand, a PC, and a processor status longword (PSL). It will dispatch to the translated handler. When the translated handler is done, it can pop the change mode operand from the emulated VAX stack frame and execute the translated instructions generated for a VAX REI.

TIE$SETEXV, in module [TIE]TIESYS, intercepts $SETEXV requests so that it can ensure a TIE$SHARE condition handler is entered to build the condition handler argument list in the form expected by a translated handler. If the request is to establish a handler rather than to remove one, TIE$SETEXV determines whether the handler is native or translated. If native, TIE$SETEXV requests the $SETEXV service, passing it the original arguments. If translated, TIE$SETEXV requests the $SETEXV service, passing it the address of its own handler, TIE$PRIMARY_HANDLER, TIE$SECONDARY_HANDLER, or TIE$LAST_CHANCE_HANDLER, depending on the system service argument VECTOR. It saves the address of the translated

vectored handler in its own data area to call whenever its own handler is called. Chapter *Condition Handling* describes condition handling and vectored handlers.

TIE$UNWIND, in module [TIE]TIESTATE, requests the native system service, passing it the arguments supplied by the translated caller, presumably a condition handler. If the service returns successfully, TIE$UNWIND determines whether the original request came from a translated condition handler. If so, TIE$UNWIND locates that condition handler's call frame and replaces the return PC in it. TIE$SHARE must emulate OpenVMS VAX $UNWIND, which replaces the condition handler call frame return PC with the address of a routine to initiate unwinding. Without this change, the condition handler would return to TIE condition handler search code, which would incorrectly look for another condition handler.

### 9.6.6 Fault-on-Execute and Interpretation of VAX Code

In addition to run-time support, TIE$SHARE provides a fallback interpreter of VAX code that was not found during translation and a feedback mechanism to provide input for a subsequent translation. The goal is that by using the feedback information to retranslate the image, interpretation will only be required for dynamically created VAX code.

As described in Section 9.5.1, when a translated image is activated, the pages containing the copy of the VAX image are modified to be fault-on-execute. Any subsequent attempt to transfer control to a location in them results in an exception. The OpenVMS fault-on-execute exception service routine signals an access violation with bit 17 set in the signal array reason mask to identify this exception. If any translated code procedures are active, they are preceded by a TIE$NATIVE_TO_TIE frame that established TIE$$N_TO_T_CH as its condition handler.

TIE$$N_TO_T_CH confirms that the exception occurred during the execution of translated code, making the following sanity checks:

- The depth in the mechanism array must be 0, indicating that the exception occurred in the same native procedure context that established it as condition handler.
- The establisher's procedure value must be that of TIE$NATIVE_TO_TIE or TIE$$N_TO_T_CH.
- The faulting virtual address must correspond to the contents of R19 at the time of the exception, indicating that the exception occurred within a translated context.
- R17 at the time of the exception must contain 0, 1, 2, or 3, namely, a valid value identifying the type of transfer.

If any check fails, TIE$$N_TO_T_CH resignals the condition.

If they all pass, it alters the signal array faulting PC and the mechanism array's saved R27 to describe the lookup routine in [TIE]TIELOOKU corresponding to the transfer type (TIE$CALLX_LOOKUP or TIE$JSB_LOOKUP, for example). It then returns the status SS$_CONTINUE, as a result of which control is transferred to that lookup routine with all other registers restored to their values at the time of the exception. The lookup routine transfers to translated code if the lookup succeeds or dispatches to the interpreter, as described in Section 9.6.2.

Chapter *Condition Handling* describes condition handling in detail.

### 9.6.7 Achieving Instruction Atomicity

In an image translated with the qualifier /PRESERVE=INSTRUCTION_ATOMICITY, some instruction atomicity is achieved because the generated code uses the intr_flag state bit, described in Section 9.2.2. Use of this flag enables detection of an interrupt or exception that has occurred in the middle of a set of native instructions emulating a single VAX instruction.

The TIE$SHARE procedure TIE$SIGC, in module [TIE]TIESTATE, uses the instruction granularity bitmap to help enforce instruction atomicity. It is called when control is transferred to a translated procedure through TIE$NATIVE_TO_TIE. In its attempt to locate the most recent VAX call frame, TIE$NATIVE_TO_TIE calls TIE$$ALIGN_VAX_STATE. If it finds a VAX call frame representing delivery of a translated AST that has interrupted execution of instructions within the translated part of the image, TIE$$ALIGN_VAX_STATE calls TIE$$IGC.

TIE$$IGC takes the following steps:

1. It identifies which translated image contains the interrupted PC. (A process may have several translated images activated at once.) TIE$$IGC determines whether that image has an instruction granularity bitmap. If not, it returns. The AST procedure can execute immediately, but atomicity is unenforceable.
2. If there is a map, it locates and tests the bit corresponding to the interrupted PC. If the bit is set, execution of that set of instructions has not yet begun, and the AST procedure can execute without further ado. TIE$$IGC returns.
3. If the bit is clear, TIE$$IGC must ensure that the set of instructions complete or appear not to have started before it allows the AST procedure to execute. It scans forward in the bitmap until it reaches a set bit representing the first instruction of the next set.

   During the scan it examines each instruction to determine whether the instruction can be interpreted safely. If the instruction, for example, modifies emulated VAX state, accesses memory other than the current call frame, is a `CALL_PAL` instruction, or is part of a load-locked/store-conditional sequence, the instruction cannot be interpreted safely.

At the end of the scan, if all instructions can be interpreted safely, TIE$$IGC interprets them, advances to the next set of instructions, and then allows the AST procedure to execute.

4. If one or more instructions cannot be interpreted safely, TIE$$IGC scans backward in the bitmap until it reaches a set bit representing the first instruction of the current set. During the scan it examines each instruction to determine whether it destroyed VAX state or whether it could be safely reexecuted.

   At the end of the scan, if all instructions can be safely reexecuted, TIE$$IGC resets the interrupted PC to the beginning of that set and allows the AST procedure to execute. (The reset is possible because at the beginning of a set of instructions emulating a VAX instruction all VAX state is recorded in a save area from which it is retrieved.)

   If all instructions cannot be safely reexecuted, TIE$$IGC signals the error TIE$_IGCNOTDONE.

## 9.7 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[LIBOTS]OTS$CALL_PROC_ALPHA.M64
[STARLET]PSIGDEF.SDL
[STARLETOLB]AMAC_CALL.MAR
[SYS]ASTDEL_STACK.M64
[SYS]SYSCHGMOD.MAR
[SYS]SYSJACKET.M64
[TIE]TIE.B64
[TIE]TIE.PREFIX
[TIE]TIE.R64
[TIE]TIEEXC.M64
[TIE]TIEIMG.M64
[TIE]TIELOOKU.M64
[TIE]TIESTATE.B64
[TIE]TIESYS.M64
[TIE]TIESYSVECT.M64

# 10 Process Deletion

. . . for dust you are and unto dust you shall return.
*Genesis* 3:19

A process can delete itself or any other process in a VMScluster system that it has the privilege to affect. For a process to be deleted, a number of cleanup actions are necessary:

- All the process's kernel threads must be terminated.
- All resources allocated to the process must be returned to the system. For example, when its virtual address space is deleted, physical memory allocated to the process is returned to the system.
- Any subprocesses of the process being deleted must be deleted.
- If the process being deleted is a subprocess, all quotas and limits taken from its parent (owner) process must be returned.
- If the creator of the process being deleted requested notification of the process's deletion through a termination mailbox, the termination message must be sent.
- Accounting information must be sent to the job controller.
- Finally, all traces of the process must be removed from the system.

Process deletion occurs in three stages: first, in the context of the process requesting the deletion; second, in the context of the initial kernel thread of the process being deleted; and finally, in system context. Figure 10.1 summarizes the actions of these stages.

## 10.1 PROCESS DELETION IN CONTEXT OF REQUESTING PROCESS

Process deletion is initiated when one process requests the Delete Process ($DELPRC) system service to delete itself or another process. If the requesting process is deleting itself, no system service arguments are required; otherwise, the requesting process can specify either the process name or the extended process ID (EPID) of the process to be deleted.

The $DELPRC system service procedure, EXE$DELPRC in module SYSDELPRC, runs in kernel mode in the context of the kernel thread requesting the system service. It performs a simple set of privilege checks and then queues a kernel mode asynchronous system trap (AST) that will cause the deletion to continue in the context of the process being deleted. Chapter *ASTs* describes the queuing and delivery of ASTs.

EXE$DELPRC performs the following steps:

1. It calls EXE$NAM_TO_PCB, in module SYSPCNTRL, to locate the process control block (PCB) of the process to be deleted.

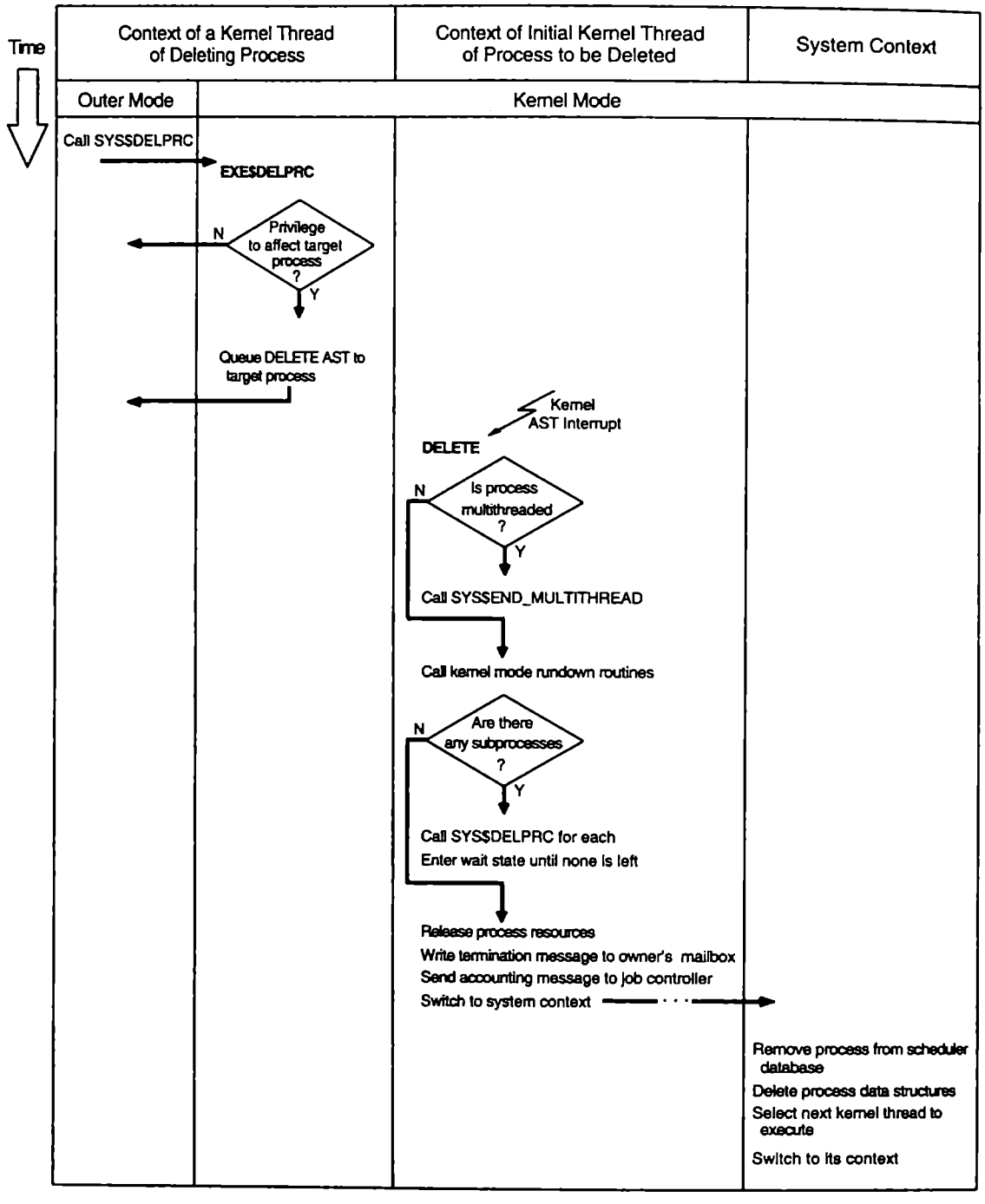



**Figure 10.1**
Main Steps in Process Deletion

EXE$NAM_TO_PCB acquires the SCHED spinlock, raising IPL to IPL$_SCHED, to synchronize its access to the scheduler database. It determines whether the input arguments specify a target process on this VMScluster node or on another node. In the former case, EXE$NAM_TO_PCB confirms the existence of the target process and the ability of the current kernel thread to delete it. Chapter 6 describes the possible relations between the two and the privileges required in each case. If the target process is identified as one on another VMScluster node, EXE$NAM_TO_PCB cannot make those checks; it can only confirm that the VMScluster node identification is valid.

- If the input arguments specified a nonexistent process on the local node or a process on a nonexistent VMScluster node, EXE$NAM_TO_PCB releases the SCHED spinlock, lowers IPL to 0, and returns an error status.

  EXE$DELPRC simply returns that error status to its requestor.
- If the input arguments specified a process on the local node that the requesting kernel thread is not allowed to delete, EXE$NAM_TO_PCB releases the SCHED spinlock, lowers IPL to 0, and returns an error status.

  EXE$DELPRC simply returns that error status to its requestor.
- If EXE$NAM_TO_PCB returns the error status SS$_REMOTE_PROC, indicating that the process may exist on another VMScluster node, EXE$DELPRC calls the clusterwide process services (CWPS) routine CWPS$P_CNTRL, in module SYSPCNTRL. CWPS$P_CNTRL transmits the deletion request to the appropriate VMScluster node and places the current kernel thread into a resource wait state (MWAIT) for resource RSN$_CLUSRV. A cooperating CWPS routine on the other node processes the request and transmits status back to this node. Through mechanisms described in Chapter 6, the waiting kernel thread is made computable, and control returns to a CWPS routine running in its context. This routine exits from the $DELPRC system service, returning the status transmitted from the other node.
- If the target process is local to this node and may be deleted by the requesting kernel thread, EXE$NAM_TO_PCB returns at IPL$_SCHED with the SCHED spinlock held. (This includes the case in which a process deletes itself.)

  EXE$DELPRC continues with the next step.

2. EXE$DELPRC tests the flag PCB$V_NODELET in the target process's PCB$L_STS. This flag is set to prevent deletion of system processes such as the swapper, NETACP, and NET$ACP. If the flag is set, EXE$DELPRC does not delete the process but instead releases the SCHED spinlock, lowers IPL, and returns the error status SS$_NODELETE. Use of

the PCB$V_NODELET flag is reserved to Digital. Any other use is unsupported.

3. If the PCB$V_NODELET flag is clear, EXE$DELPRC allocates and initializes an AST control block (ACB) to describe the kernel mode AST.
4. EXE$DELPRC marks the target process for deletion by setting the delete pending flag PCB$V_DELPEN in PCB$L_STS. If the bit was previously clear, it continues with step 5. If the bit was already set, deletion is already underway for the target process. In that event, EXE$DELPRC tests whether it is running in the context of the target process. If not, it releases the SCHED spinlock, lowers IPL, deallocates the ACB, and returns the success status SS$_NORMAL.

   If EXE$DELPRC is running in the context of the target process, that is, if the process is trying to delete itself, it tests the PCB$V_ERDACT bit in PCB$L_STS to determine whether $DELPRC was requested from an executive mode rundown routine. Chapter 8 gives information on executive mode rundown routines.

   If the bit is set, EXE$DELPRC proceeds to the next step as though the process were not already marked for deletion. Proceeding even though PCB$V_DELPEN is set ensures that process deletion will complete even if the $EXIT system service or the $DELPRC system service is rerequested by a process's executive mode rundown routine.

   If PCB$V_ERDACT is clear, EXE$DELPRC releases the SCHED spinlock, lowers IPL, deallocates the ACB, and returns the success status SS$_NORMAL.
5. If the target process is in a state of hard suspension, that is, suspended in kernel mode, EXE$DELPRC sets the process's PCB$V_RESPEN bit and reports a resume event for the process. This event is significant only for a process in scheduling state SUSP or SUSPO and causes such a process to be resumed. This mechanism is necessary because no ASTs can be delivered to a process suspended in kernel mode, including the delete process kernel mode AST. For a multithreaded target process, EXE$DELPRC reports a resume event for each of the process's kernel threads.

   If the target process is in a state of soft suspension, suspended in supervisor mode, it need not be resumed; the delete process kernel mode AST can be delivered.
6. EXE$DELPRC initializes the ACB with the process ID (PID) of the target process and the procedure value of the kernel mode AST procedure that performs the actual process deletion, procedure DELETE in module SYSDELPRC. It sets the thread-safe flag in the ACB to ensure that the AST is delivered to the initial kernel thread of a multithreaded process (see Chapter 5).
7. If EXE$DELPRC was requested from kernel or executive mode in the context of the process to be deleted, it sets the PCB$V_ERDACT bit in PCB$L_STS to disable calling executive mode rundown routines in a later step.

8. EXE$DELPRC queues a kernel mode AST to the target process's initial (or only) kernel thread, with a potential boost of PRI$_RESAVL to its software priority.

Queuing the AST to the target process's initial kernel thread makes it computable. Eventually, the scheduling code selects that kernel thread for execution.

## 10.2 PROCESS DELETION IN CONTEXT OF PROCESS BEING DELETED

Most of process deletion occurs in the context of the initial kernel thread of the process being deleted. If the kernel thread has no pending special kernel mode or other kernel mode ASTs, the DELETE AST procedure executes immediately. Note that a process whose initial kernel thread is executing or waiting at IPL 2 or above cannot be deleted because ASTs cannot be delivered to it.

Deleting a process in its own context means that its address space and process header are readily accessible. The DELETE AST procedure is therefore able to request standard system services, such as Delete Virtual Address Space ($DELTVA) and Deassign I/O Channel ($DASSGN).

Running in the initial kernel thread of the process being deleted, the DELETE AST procedure performs the following steps:

1. It first enables resource wait mode by clearing PCB$V_SSRWAIT in PCB$L_STS to minimize the chances that a service it requests will return because of insufficient resources.
2. It checks whether the process is multithreaded (whether CTL$GL_MULTITHREAD is nonzero) and, if so, requests the $END_MULTITHREAD system service (see Chapter 5). The initial kernel thread hibernates until all kernel threads other than the initial kernel thread have been deleted.
3. DELETE then searches for process-private or systemwide executive mode rundown routines to perform image-specific cleanup. Process-private executive mode rundown routines are typically part of a privileged shareable image and established by the image activator.

   If executive mode rundown routines have been declared but not yet called, DELETE sets the PCB$V_ERDACT bit in PCB$L_STS, indicating that executive mode rundown is active, and queues an executive mode AST to the process, specifying EXEC_RUNDOWN_AST as the AST procedure. DELETE then returns from the kernel mode AST procedure, leaving kernel mode and allowing the executive mode AST to be delivered.

   EXEC_RUNDOWN_AST, in module SYSDELPRC, calls whatever process-private executive mode rundown routines and systemwide executive mode rundown routines have been declared. It then requests the Change to Kernel Mode ($CMKRNL) system service to resume processing in the original DELETE code path at step 5.

4. If no executive mode rundown routines need to be called, DELETE clears the appropriate AST active bit to indicate that no kernel mode AST is active. It calls SCH$RESET_ASTSR, in module ASTDEL, to reset the AST summary register. Taking these steps enables another kernel mode AST to interrupt the DELETE AST. Although interruption of an AST by another at the same mode is usually prohibited, it may be necessary before process deletion can complete.
5. DELETE checks whether the process has an asynchronous Files-11 operation in progress. If so, it must complete before DELETE can proceed. If PCB$L_DPC is nonzero, indicating this condition, DELETE places the kernel thread into a resource wait state (MWAIT) for resource RSN$_ASTWAIT. When the queuing and delivery of a kernel mode AST end the resource wait, DELETE repeats its check. When PCB$L_DPC is zero, the DELETE procedure can continue. Chapter *ASTs* documents the field PCB$L_DPC and its use in stalling process deletion.
6. If process-private or systemwide user-specified kernel mode rundown routines exist, they are called to perform image-specific cleanup. Process-private kernel mode rundown routines are typically part of a privileged shareable image and established by the image activator.
7. DELETE then zeros each of the activated privileged library dispatch vectors that control dispatching to inner-mode routines in privileged shareable images and process-private kernel mode and executive mode rundown routines (see Chapter *System Service Dispatching*). This ensures that those rundown routines will not be called again later from the $RUNDWN system service.
8. DELETE calls EXESPRCEVT_DISPATCHER, in module LAYERED_SUBSYSTEM_ROUTINES, to notify any subsystem that requested notice of process rundown. This mechanism is intended for use by the Portable Operating System Interface (POSIX).
9. DELETE calls SYS$RMSRUNDWN to perform Record Management Services (RMS) rundown. Section 10.2.1 describes its processing.
10. If the process has any subprocesses (if its PCB$L_PRCCNT field is nonzero), they must be deleted before deletion of the owner process can continue. Section 10.4 contains an example of deleting a process with subprocesses.

    The following steps are performed to delete the subprocesses:

    a. DELETE scans the PCB vector for all PCBs whose PCB$L_OWNER field identifies this process as its creator. DELETE requests the $DELPRC system service to delete each of these subprocesses.
    b. DELETE again checks the subprocess count PCB$L_PRCCNT. If it is greater than zero, the kernel thread is placed into a resource wait state (MWAIT) for resource RSN$_ASTWAIT. It becomes computable again when the RETQUOTA special kernel mode AST returns CPU

time quota from one of the subprocesses (see step 20) and control returns to DELETE. DELETE repeats this step until the subprocess count is zero. At that point, all subprocesses have been deleted and the DELETE procedure can continue.

11. DELETE requests the Image Rundown ($RUNDWN) system service to run down the process from kernel mode and perform memory management cleanup associated with images mapped in the process-private address space (see Chapter 8). As part of the cleanup, all of P0 space and some of P1 space are deleted.
12. For each section still mapped in the process-private virtual address space, DELETE determines its virtual address region and requests the appropriate system service to delete the section's pages, $DELTVA for P1 space sections and $DELTVA_64 for P2 space sections.

    If any section pages were actually deleted by either $DELTVA or $DELTVA_64, DELETE first calls MMG$DALCSTXSCN, in module SYSCRMPSC, to free any process section table entries that no longer describe mapped pages and then requests the $RUNDWN system service once again to complete the deassignment of open channels.
13. It calls IOC$SCAN_CCB, in module IOCHANUTILS, to scan the process's channel control blocks (CCBs) to ensure that all channels have been deassigned. If any channel is still assigned, DELETE generates a fatal FILCNTNONZ bugcheck.
14. If the current process is a top-level process (if the PCB$L_OWNER field is zero), DELETE examines the job's mounted volume list. For each volume it finds that is sequential, DELETE calls MME$$DISMOU2, in module [MME]MME_DISMOU2, to notify the media management facility. For each mounted volume that DELETE finds of any type, it calls IOC$DISMOUNT, in module DISMOUNT, to dismount it. The process is temporarily granted BYPASS privilege to ensure successful completion of the dismount, in case the volume was originally mounted by a subprocess.

    If the current process is a subprocess, DELETE reassigns any volumes allocated by the subprocess to the top-level process in the job. DELETE stores the PID of that process in UCB$L_PID and sets the UCB$V_DEADMO bit in UCB$L_STS to ensure that the volume will be deallocated when it is eventually dismounted by the top-level process.
15. DELETE requests the Deallocate Device ($DALLOC) system service to deallocate all devices allocated by the process.
16. DELETE compares PCB$L_DIOLM to PCB$L_DIOCNT and PCB$L_BIOLM to PCB$L_BIOCNT. The difference between the first two fields is the number of outstanding direct I/O requests; the difference between the latter two is the number of outstanding buffered I/O requests.

DELETE spins, repeating each test, until PCB$L_DIOCNT equals PCB$L_DIOLM and PCB$L_BIOCNT equals PCB$L_BIOLM. This ensures that all outstanding process I/O requests have completed. I/O postprocessing code for completed, aborted, and canceled I/O requests is responsible for returning the PCB$L_DIOCNT and PCB$L_BIOCNT quotas to the process (see Chapter *I/O System Services*).

17. If the current process is not a subprocess, DELETE decrements one of two systemwide process counts. If the process is interactive (if PCB$V_INTER in PCB$L_STS is set), DELETE decrements the number of interactive jobs, SYS$GL_IJOBCNT. If the process is a batch job (if PCB$V_BATCH in PCB$L_STS is set), DELETE decrements the number of batch jobs, SYS$GL_BJOBCNT.
18. If the current process is a master or top-level process, DELETE deletes the jobwide logical name table.
19. DELETE clears the process name string in the PCB by zeroing the count byte. The process can no longer be identified by name.
20. If the current process is a subprocess, any remaining deductible quotas must be returned to the owner process. The following steps are taken:
    a. An I/O request packet (IRP) is allocated for use as an ACB.
    b. The procedure value of the return quota special kernel mode AST (routine RETQUOTA in module SYSDELPRC) and the PID of the owner process are stored in the ACB.
    c. The only quota that must be returned to the owner process, unused CPU time, is stored in the portion of the IRP immediately following the ACB. All other quotas are either pooled or nondeductible (see Chapter 7).
    d. Finally, the special kernel mode AST is queued to the owner process, giving it a priority boost of PRI$_RESAVL.
21. DELETE calls NSA$DEACCESS_AUDIT, in module SECURITY_AUDITING, to write any pending audits of deaccesses, for example, a close of a file whose open had been audited. It requests the $DISABLE_AUDIT system service to disable any further security auditing for the current process. It also deallocates the per-process security audit block, if one exists, to nonpaged pool.
22. If the process that created this one requested a termination mailbox message, DELETE constructs a termination message on the stack. It requests the Queue I/O Request ($QIO) system service to send the termination message to a mailbox:
    - If the process was created by a process on the same VMScluster node, DELETE sends the message to the mailbox unit specified by PCB$L_TMBU.
    - If the process was created by a process on another node (if PCB$V_NOTIFY is set in PCB$L_STS2), DELETE sends the message to the

Table 10.1 Contents of the Termination Mailbox Message Sent to the Owner Process

| *Field in Message Block* | *Size (bytes)* | *Source of Information* |
|---|---|---|
| Message type | 2 | MSG$_DELPROC[1] |
| Reserved | 2 | – |
| Final exit status | 4 | CTL$GL_FINALSTS |
| Process ID | 4 | KTB$L_EPID |
| Job ID | 4 | Not currently used |
| Logout time | 8 | EXE$GQ_SYSTIME |
| Account name | 8 | CTL$T_ACCOUNT |
| User name | 12 | CTL$T_USERNAME |
| CPU time | 4 | PHD$L_CPUTIM |
| Number of page faults | 4 | PHD$L_PAGEFLTS |
| Peak paging file usage | 4 | Not currently used |
| Peak working set size | 4 | CTL$GL_WSPEAK |
| Buffered I/O count | 4 | PHD$L_BIOCNT |
| Direct I/O count | 4 | PHD$L_DIOCNT |
| Count of mounted volumes | 4 | CTL$GL_VOLUMES |
| Login time | 8 | CTL$GQ_LOGIN |
| EPID of owner | 4 | PCB$L_EOWNER |

[1] MSG$_DELPROC is a constant indicating that this is a process termination message.

mailbox of the VMScluster server process (CLUSTER_SERVER) for it to send to the creator's node.

The message contents are listed in Table 10.1. The message size is specified by ACC$C_TERMLEN.

23. DELETE calls EXE$PRCDELMSG, in module ACCOUNT, to send an accounting message to the job controller. It sends the message unless accounting is inhibited for this process (the NOACNT flag was specified at process creation) or process termination accounting is disabled for the entire system. The contents of this message are used to fill in all relevant fields of the accounting identification and resource packets. The data structures used by the Accounting utility are described in the *OpenVMS System Manager's Manual.*
24. DELETE calls EXE$PRCEVT_DISPATCHER, in module LAYERED_SUBSYSTEM_ROUTINES, to notify any subsystem that requested notice of virtual address space deletion during process deletion. This mechanism is intended for use by the Portable Operating System Interface (POSIX).
25. After IPL is raised to 2 to prevent AST delivery, most of the remainder of P1 space is deleted. However, the P1 pages permanently locked into the working set list, the initial kernel thread's kernel stack, for example, are

not deleted. Some of P1 space, including its user stack, may have already been deleted as a result of the image reset done in step 11.

26. DELETE acquires the MMG and SCHED spinlocks to synchronize access to the memory management and scheduler databases. It releases all process-private page table pages, except those with entries for pages permanently locked into the working set, to the head of the free page list and deallocates the associated page file space. The page table pages that map pages that are permanently locked into the working set will eventually be released by process deletion code running in system context.

    Periodically, DELETE releases the MMG and SCHED spinlocks and reacquires them, so as to give other SMP members a chance to acquire them and prevent spinlock timeouts.

    When releasing page table pages, the initial kernel thread is placed into a resource wait for RSN$_MPWBUSY if a modified page table page is being written to a page file.

27. DELETE calls SCH$DELETE_CALLBACK, in module SCHEDULER. SCH$DELETE_CALLBACK executes a `CALL_PAL SWPCTX` instruction, leaving the process context of the initial kernel thread (the process being deleted) and entering system context. In system context, SCH$DELETE_CALLBACK calls DELETE_IN_SYS_CONTEXT, in module SYSDELPRC.

### 10.2.1 RMS Rundown

The DELETE kernel mode AST procedure calls SYS$RMSRUNDWN to perform RMS rundown. The service routine RMS$RMSRUNDWN, in module [RMS]RMS0RNDWN, aborts RMS I/O for the process. It transfers control to the routine RM$LAST_CHANCE, in module [RMS]RMS0LSTCH, to perform the actual rundown.

During a typical image exit, RMS performs extensive cleanup. At process deletion, however, its actions are more limited: either the process's image exited normally and RMS cleanup has already been performed, or the process may be undergoing deletion as the result of a serious error and its state and that of RMS internal data structures are suspect.

The following considerations affect RMS actions during rundown at process deletion:

- To maintain normal system operation, system resources that are still in use for files opened by the process must be deallocated.
- RM$LAST_CHANCE is only invoked at process deletion, not at normal image exit. In normal operation RMS routines execute and are synchronized in executive mode, whereas at process deletion RMS rundown routines are called from kernel mode. Complex recovery under these circumstances is not feasible.

- Recovery unit (RU) journaling is the OpenVMS mechanism provided to ensure consistency of data within a file.

  RMS RU journaling is an optional product that can be used in conjunction with RMS to ensure that user data do not become lost or inconsistent in the event of process or system failure. An application must be specifically designed to take advantage of RMS RU journaling.

  A user application that does not use this journaling is expected itself to handle potential data loss resulting from process deletion. For details on RMS journaling, consult the *RMS Journaling for OpenVMS Manual.*

RM$LAST_CHANCE scans the data structures associated with the process's open disk files to identify those eligible for cleanup. It processes "dirty" data buffers, those that have been modified but not yet written to disk. Its processing is aimed at assisting later troubleshooting or releasing system resources rather than at ensuring data integrity.

RM$LAST_CHANCE's processing of dirty buffers depends on whether the buffer is local (process-private) or global (shared). Its processing of local dirty buffers depends largely on the file organization of the data being buffered:

- It does no processing for relative or indexed files. Dirty buffers of indexed files are likely to be inconsistent and writing them to their file could cause data corruption.
- Because the structure of sequential files is simpler, it does attempt to preserve the last data records written to a sequential file, provided that the file is not marked for any type of RMS journaling.
  - For a nonshared write-accessed sequential file, it writes to disk any dirty local buffers for the file. The file is extended, if necessary, to accommodate this. It then deaccesses the file to update the RMS record attributes in the file header, particularly the end-of-file pointer.
  - For a shared write-accessed sequential file that has the deferred-write option set and whose current end-of-file is within a dirty local buffer, RM$LAST_CHANCE attempts to write that dirty buffer to disk. However, because RM$LAST_CHANCE is called from kernel mode and RMS synchronization and interlocking mechanisms are at executive mode, the writeback may not succeed.
  - No cleanup is needed for a shared write-accessed sequential file that does not have the deferred-write option set, since any modification made to a local buffer would have been immediately written back to disk.

  The attempt to preserve the last data records written to a sequential file may help a subsequent attempt to analyze process action prior to deletion. This feature is intended for problem analysis rather than for minimizing data loss.

RM$LAST_CHANCE's rundown of files with global buffers is focused on releasing system resources. It performs the following steps for any file with global buffers, regardless of file organization:

1. RM$LAST_CHANCE decrements the reference count associated with each global buffer the process was accessing to indicate one less accessor of that buffer.
2. It calls routine RM$RELEASE_GBL_BUFFERS, in module [RMS]RMSO-CLOSE, to check whether this process is the last accessor to this file's global buffer cache. If so, a cleanup is done of the global buffer section, including the release of all buffers in the global cache and the dequeuing of any global buffer locks (system-owned).

Independent of buffer type and file organization, for each file with RMS RU journaling, RM$LAST_CHANCE calls routine RM$RU_PROC_CLEANUP, in module [RMS]RMJRUREC, to perform rundown processing. This routine triggers detached RU recovery for any RU-journaled file that has open transactions at the time of the rundown. Detached recovery will roll back the effects of any such open transaction.

## 10.3 PROCESS DELETION IN SYSTEM CONTEXT

Process deletion completes in system context. DELETE_IN_SYS_CONTEXT is entered holding the MMG and SCHED spinlocks. It takes the following steps:

1. If the capability mask of the kernel thread being deleted indicates explicit affinity for particular CPUs, DELETE_IN_SYS_CONTEXT calls SCH$CLEAR_AFFINITY_BINDINGS, in module CAPABILITY, to decrement each CPU's explicit affinity count and clear KTB$L_AFFINITIES in the thread's kernel thread block (KTB), removing explicit affinity from the kernel thread. DELETE_IN_SYS_CONTEXT clears KTB$Q_CAPABILITIES to remove any user-defined capabilities.
2. DELETE_IN_SYS_CONTEXT stores the address of the null PCB in the per-CPU database field CPU$L_CURKTB and in the PCB vector slot formerly occupied by the process being deleted, thus freeing the slot for future use.
3. It increments the sequence vector element corresponding to the process index of the process being updated. If the new sequence number is larger than the maximum allowed, DELETE_IN_SYS_CONTEXT resets it to the value 1. Chapter 5 explains the use of the sequence number in validating process and kernel thread IDs.
4. The pages in process space that were permanently locked into the working set, for example, the initial kernel thread's kernel stack and the P1 pointer area, are deleted and placed at the head of the free page list. The

process header pages and the L1 page table page will be deleted by the swapper when the process header is deleted.

5. DELETE_IN_SYS_CONTEXT removes each remaining ACB from each of the ACB queues in the KTB and the PCB. It deallocates each to nonpaged pool unless its ACB$V_NODELETE bit is set. If the ACB$V_NODELETE bit is set, the ACB is assumed to be part of another data structure whose deletion is not allowed.
6. DELETE_IN_SYS_CONTEXT removes any CWPSSRV data structures, which describe CWPS requests in progress, from the PCB$Q_CWPSSRV_QUEUE queue of the process being deleted. It inserts them on the swapper's PCB$Q_CWPSSRV_QUEUE queue.

   These structures cannot be deleted until the stalled fork thread that expects to access them is resumed by the arrival of a response from another VMScluster node. When the response arrives, the fork thread determines that the requestor process was deleted and deallocates the structures.

   DELETE_IN_SYS_CONTEXT scans the swapper's PCB$Q_CWPSSRV_QUEUE for any CWPSSRVs that describe remote process creation requests that completed after the creator process was deleted. If it finds any, it removes them from the queue and deallocates them to nonpaged pool.

   Chapter 6 contains more information on CWPS requests and CWPSSRVs.
7. If the process had an extended rights list, it is deallocated to nonpaged pool.
8. The process count field in the job information block (JIB) is decremented in an interlocked manner. If the process being deleted is a top-level process (the PID of the process being deleted is equal to the master PID field in the JIB), the JIB is deallocated.
9. If the process being deleted is a subprocess, its owner's subprocess count, PCB$L_PRCCNT, is decremented. If the owner process is also being deleted, its initial kernel thread is currently in a wait state, waiting for the contents of this field to become zero. DELETE_IN_SYS_CONTEXT declares the resource RSN$_ASTWAIT available to make the owner process's initial kernel thread computable so that it can check the value of PCB$L_PRCCNT. If the value is now zero, the owner can continue with its own deletion.
10. DELETE_IN_SYS_CONTEXT deallocates the PCB to nonpaged pool.
11. It decrements SWP$GW_BALCNT, the number of processes in the balance set.
12. It tests whether the header reference count for this PHD slot is nonzero and, if so, generates the fatal bugcheck INCONMMGST. A nonzero reference count would be a serious inconsistency, indicating that process-private page table pages other than the level 1 page table still exist.

13. It calls SCH$SWPWAKE to awaken the swapper because there is a process header to be removed from the balance set slot area (see Chapter *The Swapper*).
14. DELETE_IN_SYS_CONTEXT stores the value −1 into the process index array entry for the process header of the process being deleted and increments SCH$GW_DELPHDCT. As a result of these actions, the swapper, when it next executes, will delete the process header.
15. DELETE_IN_SYS_CONTEXT decrements SCH$GW_PROCCNT, the systemwide process count.
16. Finally, DELETE_IN_SYS_CONTEXT releases the MMG spinlock and returns to SCH$DELETE_CALLBACK still holding the SCHED spinlock. SCH$DELETE_CALLBACK takes the following steps:

    a. It gets the priority of the initial kernel thread of the process being deleted from the per-CPU database.
    b. It clears the bit representing the CPU's ID in the longword corresponding to the priority in the array at SCH$AL_CPU_PRIORITY.
    c. If there are no other kernel threads at this priority current on any CPU, it clears the bit corresponding to that priority in SCH$GQ_ACTIVE_PRIORITY.
    d. It proceeds to select the next kernel thread to run (see Chapter 2).

## 10.4 DELETION OF A PROCESS THAT OWNS SUBPROCESSES

In early versions of VAX VMS (from which OpenVMS Alpha derives) prior to the existence of the JIB and its jobwide pooled quotas (see Chapter 7), several quotas were charged against a process when it created a subprocess. At deletion of the subprocess, the subprocess returned those quotas. For a subprocess being deleted to return correct quotas to its creator, the subprocess must wait until all the subprocesses it created have returned their quotas. Consequently, the deletion of a process is delayed until all its subprocesses are deleted.

All the quotas treated in this way are now pooled in the JIB except for CPU time limit, which is the only quota returned at subprocess deletion.

During the execution of the DELETE AST procedure, a check is made to see if the process being deleted owns any subprocesses. If it does, these subprocesses must be located and deleted first.

There are, however, no forward pointers in the JIB or PCB of an owner process to indicate which subprocesses it has created. The only indication that a process has created subprocesses is a nonzero value in PCB$L_PRCCNT. The process's subprocesses can only be located by scanning all the PCBs in the system until each PCB is located whose owner field contains the PID of interest.

The details of this situation can best be illustrated with an example. Figure 10.2 shows a process whose process ID equals 10035 and whose name is

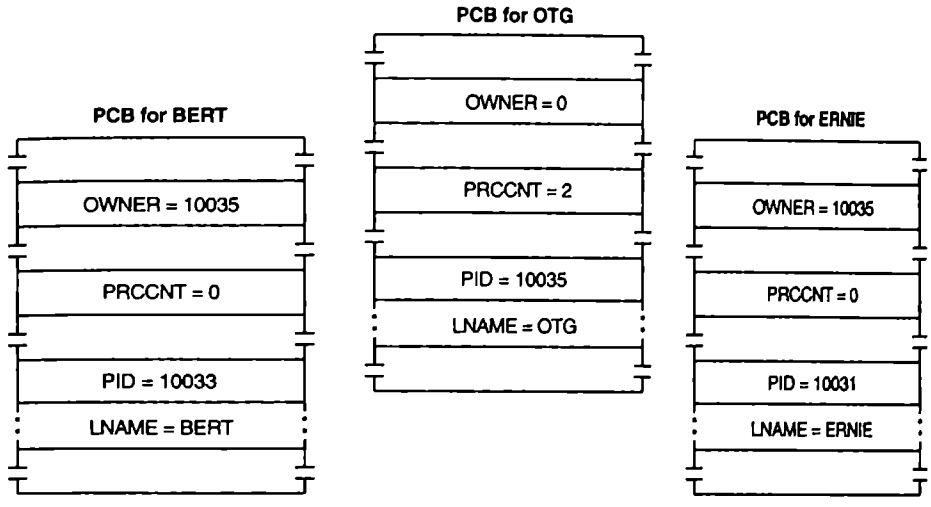



**Figure 10.2**
Sample Job to Illustrate Process Deletion with Subprocesses

OTG. The process OTG owns two subprocesses: the first has a process ID of 10033 and the name BERT; the second has a process ID of 10031 and the name ERNIE.

Neither of these subprocesses owns any further subprocesses. The following steps occur when OTG is being deleted.

Assume that the processes' priorities are such that the processes execute in the order OTG, BERT, and ERNIE.

1. The deletion of process OTG proceeds normally until the DELETE AST procedure determines that OTG has created two subprocesses. It scans the PCB vector and locates the two PCBs containing 10035 in the PCB$L_OWNER field and requests the $DELPRC system service for each process. As a result, the DELETE kernel mode AST is queued to the two subprocesses and their initial kernel threads are made computable. Process OTG's kernel thread is placed into MWAIT for resource RSN$_ASTWAIT because the process's count of owned subprocesses is nonzero (actually 2, at this point).
2. Given the assumed priorities in this example, process BERT's initial kernel thread executes next. Its deletion proceeds past the point where process OTG stopped because BERT owns no subprocesses. However, the next step in the DELETE AST procedure determines that process BERT is a subprocess and must return quotas to its owner. The return of quotas is accomplished by queuing a special kernel mode AST (RETQUOTA) to process OTG, changing its state back to computable. Although OTG might run again as a result of the state change, it would reenter its wait because its PCB$L_PRCCNT is still nonzero.

When the system context portion of the deletion of BERT has finished with all actions that require the presence of the JIB, DELETE_IN_SYS_CONTEXT decrements OTG's owned subprocess count and declares the resource RSN$_ASTWAIT available, awakening OTG's initial kernel thread. However, OTG's count of owned subprocesses is still not zero (down to 1 now), so process OTG's initial kernel thread is put back into a resource wait. The deletion of process BERT continues until BERT is entirely deleted.

3. Process ERNIE's initial kernel thread now begins to execute the DELETE AST procedure. It has no owned subprocesses, but it is a subprocess. Another RETQUOTA AST is queued to process OTG, and OTG's owned subprocess count decremented (finally to zero).
4. When process OTG's initial kernel thread resumes execution, the DELETE AST procedure finds that the owned subprocess count is zero and continues with the deletion of OTG.

## 10.5 RELEVANT SOURCE MODULE

The source module described in this chapter is

[SYS]SYSDELPRC.MAR

# 11 Process Dynamics

In my end is my beginning.
Motto of Mary, Queen of Scots

Other chapters in this volume describe process creation, image activation, and process deletion. This chapter summarizes the flow through these components for a process that executes a single image.

This chapter also describes the manner in which OpenVMS components create processes on a user's behalf. It examines the circumstances in which the various components are invoked and the resulting process types.

In addition, this chapter describes the mechanisms supporting the most common kind of process, one that executes several images consecutively. Because this mode of operation occurs in all interactive and batch processes, these two process types are discussed in detail.

## 11.1 OVERVIEW

A process can be classified by several characteristics:

- It is either a subprocess and part of its creator's job tree, sharing its job information block (JIB), or it is detached from its creator, a top-level process with an independent job tree of its own.
- It either interacts with a user and receives input from a terminal, or it is noninteractive and receives input from a file or device.
- It includes a command language interpreter (CLI) and can make the transition from one image to another, or it executes only one image and exits when the image does.

Various OpenVMS components initiate process creation by requesting the Create Process ($CREPRC) system service. They include

- The job controller for interactive and batch processes
- The Digital command language (DCL) CLI for subprocesses and noninteractive processes
- NETACP and NET$ACP, respectively, for DECnet Phase IV and Phase V network processes

Arguments to the $CREPRC system service determine process characteristics, particularly the arguments UIC, STSFLG, INPUT, and IMAGE. Chapter 7 discusses this system service and its arguments in detail. Tables 11.1 and 11.2 provide examples of arguments passed to the $CREPRC system service by OpenVMS components.

Through the IMAGE argument, the creator specifies the main image to be activated in the new process. If the image is anything other than LOGINOUT, the process will not include a CLI. Such a process can execute only a single main image. When the image exits, the process is deleted. Section 11.2 describes the overall flow from creation to deletion within a process without a CLI.

Typically, however, the first main image to run in a process is LOGINOUT. One of its main functions is to map a CLI into the process's P1 space.

A CLI enables the process to execute successive images, accomplishing the transition from one image to the next. This mode of operation occurs in all interactive and batch processes, and is optional but common for detached and network processes. Sections 11.4 and 11.5 provide more information on LOGINOUT and CLIs. A complete description of either, however, is beyond the scope of this chapter. Section 11.7 summarizes the flow from one image to the next in a process with a CLI.

Although OpenVMS Alpha Version 7.0 supplies only the DCL CLI, the operating system can deal with multiple CLIs and the description in this chapter is based on that capability.

Section 11.8 describes LOGINOUT's role in logging a process out.

## 11.2 SINGLE-IMAGE PROCESS

A process is created to execute a single specified image when code in another process requests the $CREPRC system service passing that image name. When the image exits, the process is deleted.

Chapter 7 describes the $CREPRC system service and its arguments. It also describes the code that executes in the context of the new process: first, SWP$SHELL_INIT, in module SHELL*x*K and then, EXE$PROCSTRT, in module PROCSTRT. Figure 11.1 summarizes the flow of system code within such a single-image process, beginning with EXE$PROCSTRT.

The numbers in the following list correspond to numbered steps in Figure 11.1:

① EXE$PROCSTRT begins execution in kernel mode. After preparing the process to activate an image, it executes a `CALL_PAL REI` instruction to leave kernel mode and enter executive mode for the next two steps.

② Running in executive mode, EXE$PROCSTRT requests the Image Activator ($IMGACT) system service, an executive mode system service, to activate the image specified as a $CREPRC argument and any shareable images with which it was linked.

③ It declares an executive mode exit handler, EXE$RMSEXH, to close files left open when the image exits.

④ EXE$PROCSTRT executes a `CALL_PAL REI` instruction to leave executive mode and enter user mode to execute the next steps.

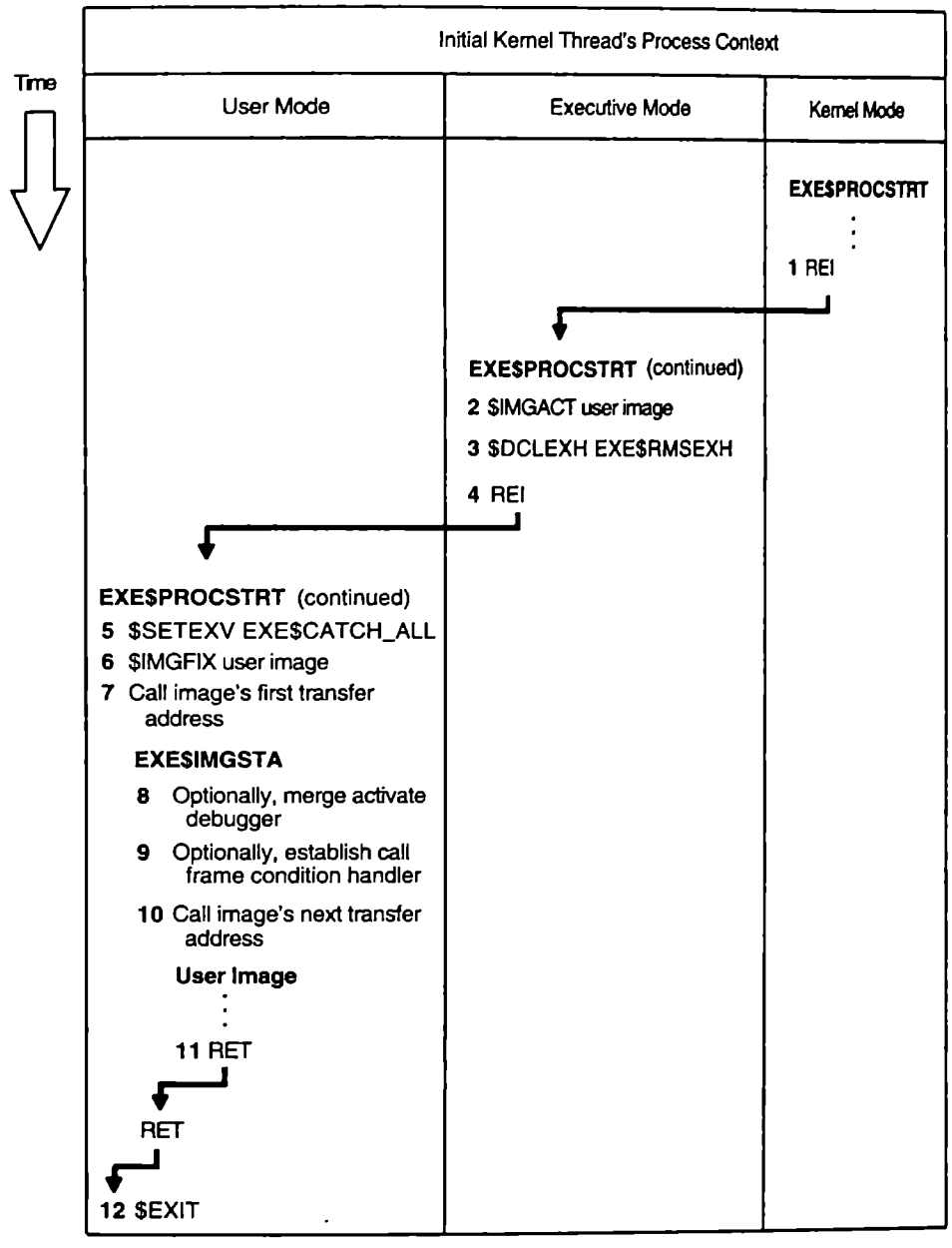



**Figure 11.1**
Example of Process That Executes a Single Image—Part 1

⑤ It establishes EXE$CATCH_ALL as the last chance condition hander (see Chapter *Condition Handling*) for user mode.

⑥ It requests the Address Relocation Fixup ($IMGFIX) system service to perform relocations and fixups for the images activated in step 2.

⑦ It calls the procedure specified by the image's first transfer address (see Chapter 8), which is typically the Image Startup ($IMGSTA) system service.

⑧ EXE$IMGSTA, in module SYSIMGSTA (see Chapter 8), optionally activates the debugger, based on image link and $CREPRC flags.

⑨ If it does not activate the debugger, EXE$IMGSTA establishes a condition handler for the current procedure frame to get control on signals the image does not handle. After gaining control, the condition handler would invoke the debugger, invoke the traceback handler, or resignal for EXE$CATCH_ALL to handle it.

⑩ EXE$IMGSTA calls the procedure specified by the image's next transfer address.

Each transfer address except the last, which is the entry point for the main image, represents a procedure that performs some initialization and then calls the next transfer address. Eventually the main image is entered and begins execution.

When the main image finishes its task, it can exit in one of several ways:

- As shown in the figure, it can execute a RET instruction, returning control back through the call chain of nested initialization routines to EXE$PROCSTRT, which requests the Exit ($EXIT) system service.
- It can directly request the $EXIT system service.
- It can call a language-specific run-time library procedure that requests the $EXIT system service.

Figure 11.2 shows the flow of control in such a process when it requests the $EXIT system service. The numbers in the following list correspond to numbered steps in Figure 11.2:

① The $EXIT system service (see Chapter 8) is a mode of caller service. When requested from a multithreaded application, EXE$EXIT, in module SYSEXIT, terminates multithreading in the process and ensures that $EXIT is executing in the process's initial kernel thread. EXE$EXIT then requests the $EXIT_INT system service, a kernel mode system service.

② EXE$EXIT_INT, in module SYSEXIT, determines the access mode from which it was requested.

③ It tests whether any exit handlers have been declared for the requestor's mode, in this case user mode. In this example, the user image declared at least one exit handler. EXE$EXIT_INT cleans up the kernel stack and executes a CALL_PAL REI instruction to return to user mode at local routine EXIT_CALLBACK.

④ EXIT_CALLBACK calls each user mode exit handler that the image and its shareable images declared.

⑤ To return to kernel mode, EXIT_CALLBACK requests the $EXIT system service. (The control flow in the figure is slightly simplified in that it omits EXE$EXIT's request of $EXIT_INT and reentry into EXE$EXIT_INT.)

⑥ EXE$EXIT_INT next checks whether any supervisor mode exit handlers have been declared. In this example, none were declared.

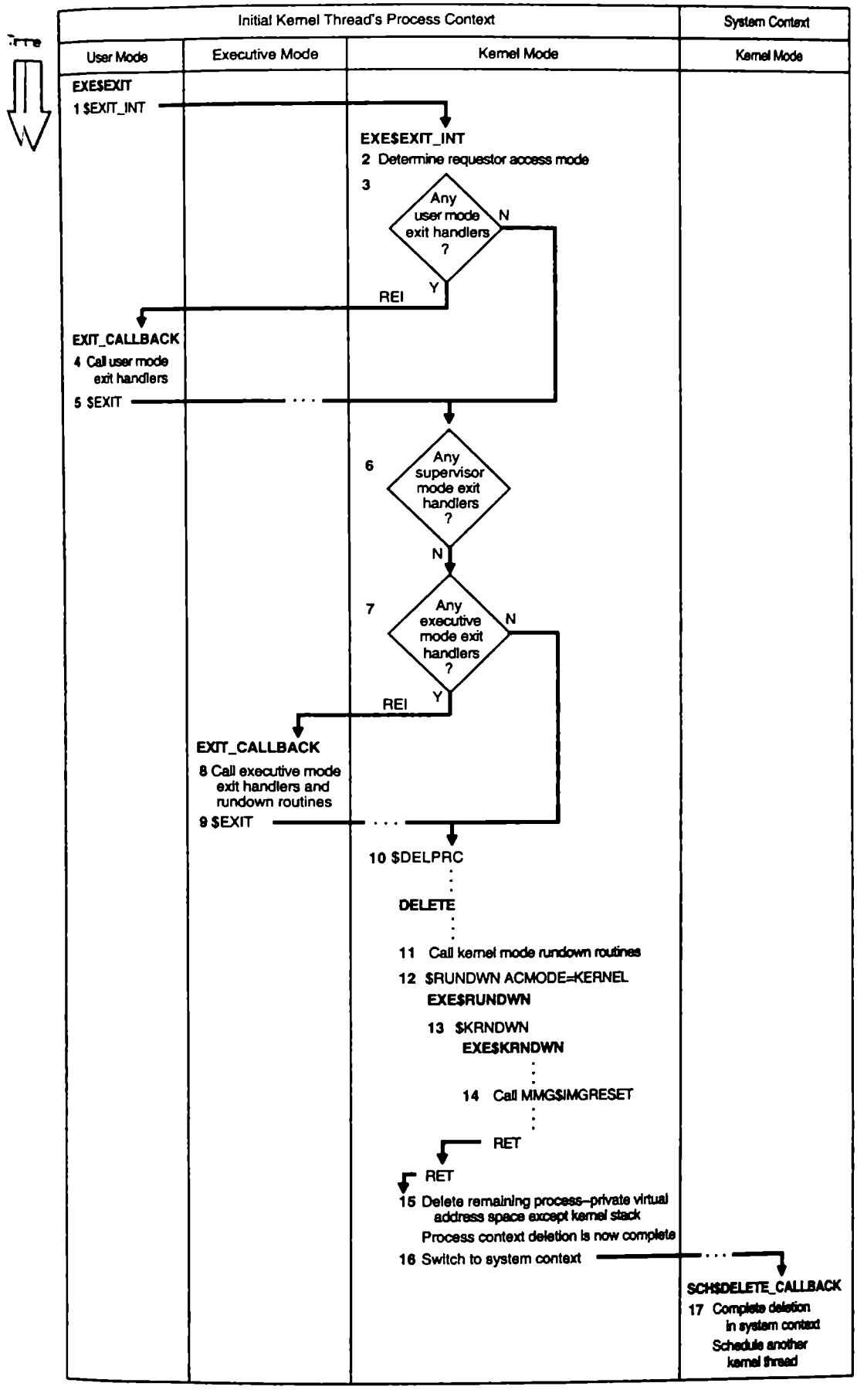



**Figure 11.2**
Example of Process That Executes a Single Image—Part 2

⑦ It checks whether any executive mode exit handlers have been declared. At least one executive mode exit handler is declared in all processes, EXE$RMSEXH. EXE$EXIT_INT cleans up the kernel stack and leaves kernel mode.

⑧ From executive mode, EXIT_CALLBACK calls each executive mode exit handler. The actions of EXE$RMSEXH are described in Chapters 8 and 10. EXIT_CALLBACK also calls any executive mode rundown routines that have been established and sets bit PCB$V_ERDACT in PCB$L_STS to indicate they have been called.

⑨ To return to kernel mode, EXIT_CALLBACK requests the $EXIT system service, which requests $EXIT_INT.

⑩ EXE$EXIT_INT requests the Delete Process ($DELPRC) system service to delete itself. EXE$DELPRC, in module SYSDELPRC (see Chapter 10), queues a normal kernel mode asynchronous system trap (AST) to the process being deleted (in this case, the current process).

⑪ DELETE, the kernel mode AST procedure, calls kernel mode rundown routines. It then resets the activated privileged library dispatch vector (see Chapter *System Service Dispatching*) to clear all established rundown routines.

⑫ It then requests the Image Rundown ($RUNDWN) system service with an argument of kernel mode. $RUNDWN is a mode of caller service (see Chapter 8).

⑬ Requested from kernel mode, EXE$RUNDWN, in module SYSRUNDWN, immediately requests the Image Rundown ($KRNDWN) system service.

⑭ EXE$KRNDWN, in module SYSRUNDWN, cleans up process activity, such as queued locks, allocated devices, and outstanding I/O, and then calls MMG$IMGRESET, in module PHDUTL.

⑮ MMG$IMGRESET (see Chapter 8) deletes the nonpermanent parts of process-private address space.

It returns to its caller, which returns to its caller, and so on.

⑯ Control returns to DELETE, which cleans up additional process activity, sends accounting information to the job controller, and deletes all the remaining process-private address space except the kernel stack, on which it is running.

⑰ It calls SCH$DELETE_CALLBACK, in module SCHEDULER, which switches to the context of the system hardware privileged context block (HWPCB) and completes the deletion of the process from system context.

It then tries to schedule another kernel thread to run.

More commonly, a process is created to run a series of images. The sections that follow describe more typical process creations and the effect of mapping a command language interpreter within the process. Section 11.7 summarizes the flow within such a process.

## 11.3 THE JOB CONTROLLER AND PROCESS CREATION

The job controller process manages the creation of nearly all interactive and batch processes. It creates an interactive process in response to unsolicited terminal input and a batch process as a result of the CLI response to the SUBMIT command.

The job controller is notified by the terminal class driver of unsolicited input via the job controller mailbox. The job controller is requested by the queue manager to create a batch process. In response to each of these, the job controller creates an appropriate process.

The process created by the job controller executes the image LOGINOUT. LOGINOUT's actions, especially mapping a CLI into P1 space, differentiate processes that can execute multiple images in succession, such as interactive and batch processes, from processes that exit after the execution of a single image.

### 11.3.1 Unsolicited Terminal Input

The term *unsolicited terminal input* refers to characters entered on a device that has no outstanding read request. The common terminal driver character-processing routine, TTY$PUTNEXTCHAR, in module [TTDRVR]TTYCHARI, handles unsolicited terminal input on an unowned terminal roughly as follows:

- If the terminal has the characteristic NOTYPE_AHEAD, the driver ignores the unsolicited input and dismisses the interrupt. This characteristic means that no interactive login is possible on that terminal.
- If the terminal has the AUTOBAUD characteristic, the driver tests the incoming character, senses the baud rate, and sets it as appropriate.

  If the character is a standard terminator recognized by the driver (for example, carriage return), the driver sends a message to the job controller mailbox, notifying the job controller that an unowned terminal has received an unsolicited interrupt. Typically, because a type-ahead buffer has not yet been allocated for the terminal, the driver initiates a fork thread to create one.
- If the terminal has the SECURE characteristic and the driver receives a break signal, the driver sends an unsolicited interrupt message to the job controller mailbox. Typically, the driver initiates a fork thread to create a type-ahead buffer.

In a sense, the job controller is the default owner of all otherwise unclaimed terminals.

The job controller routine that responds to unsolicited terminal input, UNSOLICITED_INPUT in module [JOBCTL]UNSOLICIT, simply requests the $CREPRC system service. Table 11.1 shows the arguments it passes to the system service.

Table 11.1 Arguments Resulting in Interactive Process Creation

| *Argument Passed to $CREPRC* | *Value* |
|---|---|
| Process name | *_ttcu:* |
| User identification code | [1,4] |
| Image name | SYS$SYSTEM:LOGINOUT.EXE |
| SYS$INPUT | *ttcu:* |
| SYS$OUTPUT | *ttcu:* |
| SYS$ERROR | *ttcu:* |
| Base priority | DEFPRI (SYSGEN parameter) |
| Privilege mask | TMPMBX, NETMBX, SETPRV |
| Status flags | PRC$V_INTER |

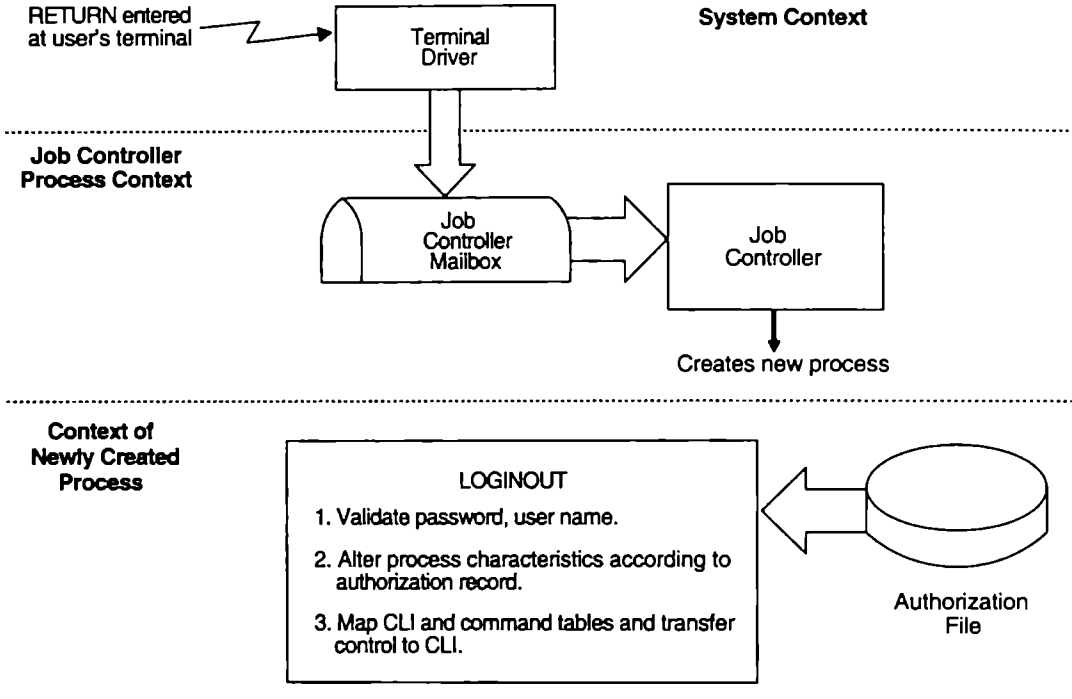

RETURN entered at user's terminal
Terminal Driver
System Context
Job Controller Process Context
Job Controller Mailbox
Job Controller
Creates new process
Context of Newly Created Process
LOGINOUT
1. Validate password, user name.
2. Alter process characteristics according to authorization record.
3. Map CLI and command tables and transfer control to CLI.
Authorization File

**Figure 11.3**
Creation of an Interactive Process

The string *ttcu:* indicates the controller and unit of the terminal where the unsolicited input was typed. For example, the terminal device type can be an actual physical device; an LT device, if the terminal is connected through a DECserver; an RT device, if the terminal is remote; a VT device, if virtual terminal support is enabled; or an FT device for DECwindows.

The job controller creates each interactive process with a process name indicating its input device and LOGINOUT as the image to be executed. The creation of an interactive process is shown in Figure 11.3.

### 11.3.2 SUBMIT Command

When the SUBMIT command is entered, DCL activates the SUBMIT.EXE image. SUBMIT sends a message to the queue manager by requesting the Send to Job Controller and Wait ($SNDJBCW) system service, which uses the interprocess communication ancillary control process.

The queue manager receives the message and creates a job record in its in-memory database. It inserts the job record into an internal list of pending requests for the desired batch execution queue or generic queue. When the number of active jobs in a batch execution queue drops below the maximum value, the queue manager selects the queue's highest priority pending request. It sends an interprocess communication (IPC) message to the job controller and changes the state of that job to starting.

The job controller receives the IPC message and requests the $CREPRC system service to create a process for that request, specifying LOGINOUT as the image to be executed.

Table 11.2 shows the arguments that the job controller passes to $CREPRC for a batch process.

LOGINOUT requests the $SNDJBCW system service to get more information about the batch process, for example, the name of its CLI, its priority, and its working set list quotas. In response to this request, the queue manager changes the job state from starting to executing and returns the requested information. LOGINOUT redefines SYS$INPUT to be the name of the batch command procedure, and SYS$OUTPUT and SYS$ERROR to be the names of a log file in an appropriate directory. Because LOGINOUT maps the appropriate CLI into the process's P1 space, the batch input file can contain a series of command language statements.

Figure 11.4 shows the processing of the SUBMIT command.

Table 11.2 Arguments Resulting in Batch Process Creation

| *Argument Passed to $CREPRC* | *Value* |
|---|---|
| Process name | BATCH_*nnn*[1] |
| User Identification Code | [1,4] |
| Image name | SYS$SYSTEM:LOGINOUT.EXE |
| SYS$INPUT | QMAN*xxxxxxxxyyyyyyyyqueue-manager-name*[2] |
| SYS$OUTPUT | _NLA0: |
| SYS$ERROR | _NLA0: |
| Base priority | DEFPRI (SYSGEN parameter) |
| Privilege mask | All |
| Status flags | PRC$V_BATCH |

[1] *nnn* is the queue entry number of the job.

[2] *xxxxxxxx* is the eight-digit queue entry number. *yyyyyyyy* is the eight-digit incarnation number of the queue manager. *queue-manager-name* is the name of the queue manager, which is SYS$QUEUE_MANAGER by default.

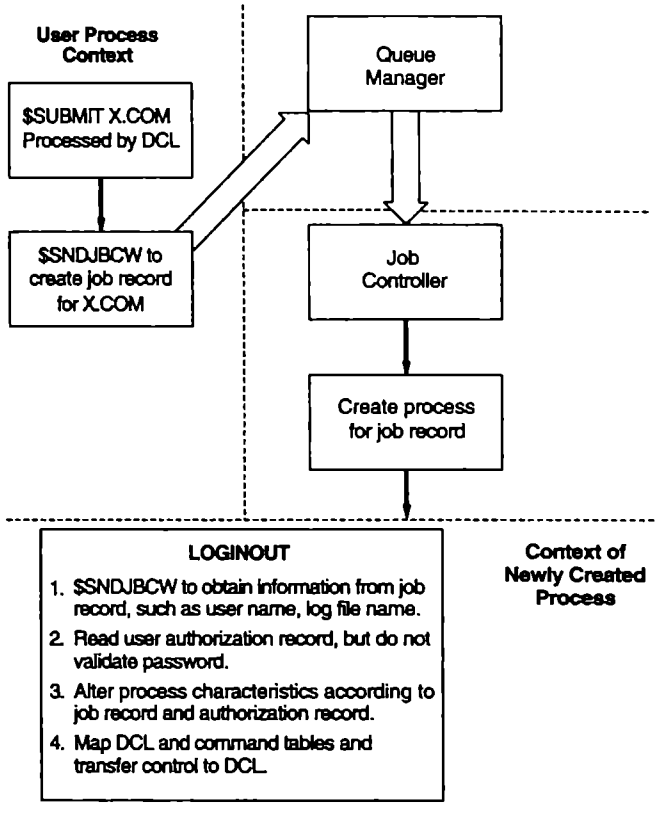

User Process Context
$SUBMIT X.COM Processed by DCL
$SNDJBCW to create job record for X.COM
Queue Manager
Job Controller
Create process for job record
LOGINOUT
1. $SNDJBCW to obtain information from job record, such as user name, log file name.
2. Read user authorization record, but do not validate password.
3. Alter process characteristics according to job record and authorization record.
4. Map DCL and command tables and transfer control to DCL.
Context of Newly Created Process

**Figure 11.4**
Creation of a Batch Process

## 11.4 THE LOGINOUT IMAGE

The LOGINOUT image has three major functions when run as the first image in a process:

1. It validates a user's access to the system, checking password information in the authorization file.
2. It adjusts various process quotas and defaults based on information from the authorization file or from the job controller.
3. It maps a CLI into P1 space.

In addition, LOGINOUT performs batch job step initialization and logout (see Section 11.8).

LOGINOUT need not perform all these functions for every process. Its actions are based on the original arguments passed to the $CREPRC system service, stored in the process control block (PCB), process header (PHD), and P1 space. For example, it does not perform password validation or read the authorization file record if the $CREPRC STSFLG argument PRC$V_NOUAF was specified.

The LOGINOUT image is installed with privileges, which it enables and disables based on the current function. The image executes primarily in user mode, with some executive and kernel mode procedures.

Normally, LOGINOUT is activated because a $CREPRC IMAGE argument specifies LOGINOUT and the image is activated by EXE$PROCSTRT. However, under certain conditions, the image activator independently invokes LOGINOUT (see Chapter 8).

OpenVMS Alpha LOGINOUT includes the capability to call site-specific routines (LOGINOUT callout routines). These routines support customer login security programs such as smart card programs, pocket authenticator programs, and other alternative identification and authentication programs. They enable sites to combine portions of LOGINOUT's security policy functions with site security login functions to establish a customized login security environment.

A positive value for the SYSGEN parameter LGI_CALLOUTS enables LOGINOUT installation callouts. The value defines the number of site-specific shareable images that contain LOGINOUT callout routines. The system manager defines the executive mode logical name LGI$LOGINOUT_CALLOUTS in the system table with equivalence names that identify these shareable images. Each shareable image defines the universal symbol LGI$LOGIN_CALLOUTS as the name associated with a table of callout routines. When LOGINOUT runs, it activates each such shareable image and records the value of LGI$LOGIN_CALLOUTS within that image. At particular points in LOGINOUT's execution, it calls the corresponding callout routine in each of these shareable images. If one of these routines returns an error status, LOGINOUT aborts.

The *OpenVMS Utility Routines Manual* provides details on the creation of LOGINOUT callout images and the interfaces to callout routines.

### 11.4.1 LOGINOUT and Interactive Processes

When the LOGINOUT image executes in an interactive process created in response to unsolicited terminal input, it must verify that the user has access to the system before proceeding with the rest of its operations. Beginning at routine START, in module [LOGIN]LOGIN, it performs the following steps:

1. It establishes a user mode procedure frame condition handler to service any exceptions or software conditions that occur while LOGINOUT is executing. Should this handler be called, it first requests the Put Message ($PUTMSG) system service to write an error message. It then checks the type and severity of the condition. If the status code has not already been stored in P1 space, the handler stores it in preparation for writing the code to the termination mailbox.

If the condition is a severe error and the failed login has not yet been audited, the handler calls LGI$SECURITY_AUDIT, in module [LOGIN]AUDIT, to audit the failure.

If the condition is a severe error, the handler calls any LGI$ICR_FINISH LOGINOUT callout routines and requests the $EXIT system service from executive mode, causing the process to be deleted. Otherwise, it returns, and LOGINOUT continues execution.

LOGINOUT declares this same condition handler for many of its executive mode procedures.

2. LOGINOUT requests the Get Job/Process Information and Wait ($GETJPIW) system service to obtain the user name, process status flags, job type, and process owner.
3. LOGINOUT requests the Get Device Information and Wait ($GETDVIW) system service to obtain the name and characteristics of SYS$INPUT.
4. If LOGINOUT callouts are enabled on this system, LOGINOUT tries to translate LGI$LOGINOUT_CALLOUTS as an executive mode logical name in the system logical name table. For LOGINOUT callouts to be made, there must be a definition with LGI_CALLOUTS equivalence names. It activates an image for each equivalence name and records the address of LGI$LOGIN_CALLOUTS within that image.
5. Continuing in routine LOGIN, in module [LOGIN]LOGIN, LOGINOUT requests the Set Privileges ($SETPRV) system service to acquire necessary privileges temporarily.
6. It temporarily changes the protection of the process-permanent data (PPD) region in P1 space to make it writable from user mode. This enables much of LOGINOUT to execute in user mode. This region is shared by LOGINOUT and the CLI it maps.
7. LOGINOUT translates the logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR in the LNM$PROCESS table and saves the resultant strings for later use.
8. LOGINOUT initializes the PPD region in P1 space and records at PPD$T_FILENAME the equivalence name for SYS$INPUT.
9. It initializes the argument vector for LOGINOUT callout routines and calls each image's LGI$ICR_INIT routine.
10. LOGINOUT classifies the process as one of the following five mutually exclusive types based on the following criteria:

    - Batch—The batch bit is set in CTL$GL_CREPRC_FLAGS, a copy of the flags specified to the $CREPRC system service.
    - Network—The network bit is set in CTL$GL_CREPRC_FLAGS.
    - Subprocess—The parent PID is nonzero.
    - Interactive—The interactive bit is set and the no-password bit is clear in CTL$GL_CREPRC_FLAGS.

LOGINOUT further subdivides interactive processes into DECwindows processes and all others. A DECwindows process is an interactive process whose input device class is DC$_WORKSTATION.

- Detached—Anything not covered by the previous types.

It performs type-specific initialization.

11. For an interactive process, typically one created in response to unsolicited input from a terminal, LOGINOUT performs the following steps largely in routines INIT_INTERACTIVE and INTERACTIVE_VALIDATION, both in module [LOGIN]INTERACT, and in routine INIT_USER, in module [LOGIN]INITUSER.

    a. It initializes the user name and account name fields in the JIB and P1 space to the string `<login>`. The user name is reset at step 11n. The account name is reset at step 34.
    b. It creates process-permanent files for the input and output devices through calls to Record Management Services (RMS). LOGINOUT redefines the logical names SYS$INPUT and SYS$OUTPUT in the LNM$PROCESS table. It defines the logical names SYS$ERROR and SYS$COMMAND with the same equivalence strings as SYS$OUTPUT and SYS$INPUT. It prefixes the equivalence names for these logical names by four bytes: an escape ($1B_{16}$), a null character ($00_{16}$), and the two-byte internal file identifier (IFI) returned by RMS. When RMS receives such a string as a result of logical name translation, it uses the IFI as an index into one of its internal tables. Accessing by IFI allows fast access to these commonly used files.
    c. In the case of an interactive login, the input device must be a terminal device. Otherwise, LOGINOUT signals the error status LGI$_INVINPUT and exits.
    d. If the terminal line has modem control enabled, LOGINOUT requires the TT$V_REMOTE bit to be set. This bit notifies the driver that the process must be logged off or disconnected if the modem signals disappear.
    e. LOGINOUT determines whether the job type is local, dialup, or remote, based on the characteristics of the SYS$INPUT terminal. Except for interactive DECwindows processes, it stores the terminal name in PCB$T_TERMINAL.

        It marks an interactive DECwindows process as local but does not store a terminal name for it.
    f. If LOGINOUT callouts are enabled, it initializes callout variables specific to interactive jobs and calls each image's LGI$ICR_IACT_START routine.
    g. LOGINOUT determines whether there is a system password and whether it applies to this terminal. If there is, it issues a timed, no-echo read to the terminal and checks the password entered by the user.

h. It then translates the logical name SYS$ANNOUNCE and writes the announcement message defined by the system manager.

i. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_IDENTIFY routine.

j. LOGINOUT checks whether autologins are enabled for the terminal that is logging in. If they are, LOGINOUT looks up the terminal name in SYS$SYSTEM:SYSALF.DAT to determine the user name associated with the terminal. It then reads the user authorization file (UAF) record associated with the user and stores the user name in the JIB and in CTL$T_USERNAME in P1 space.

k. If autologins are not enabled for the SYS$INPUT terminal, LOGINOUT prompts on it for the user name. It reads and parses the input, noting the presence of qualifiers, such as /CONNECT and /CLI. It opens the system authorization file and reads the record associated with that user, if any.

l. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_AUTHENTICATE routine.

m. Whether the desired UAF record exists or not, LOGINOUT always prompts for the password, thus providing no information about the existence of the account in case of attempted break-in. It reads and verifies the password and, if there is a secondary password for the account, prompts for, reads, and verifies that as well.

n. LOGINOUT stores the user name in the JIB and in CTL$T_USERNAME.

o. If the account is captive or restricted, LOGINOUT checks that the user did not include login qualifiers to change aspects of the process environment fixed for that account.

p. LOGINOUT then performs a scan of the intrusion database in nonpaged pool. The type of scan performed depends on the success of user validation.

   If a user validation error (such as invalid user name or password) has occurred, a suspect scan is performed. If evasion is in effect, the user name is set and a break-in audit is performed. Otherwise, the failed password count is incremented in the user's UAF record, and a corresponding intrusion record is either created or updated.

   If the login was valid, an intruder scan is performed. If the user is found to be an intruder, a break-in audit is performed and the login terminates.

12. LOGINOUT resets some of the process attributes using information extracted from the authorization file and overriding the attributes established when the process was created:

    - Logical name definition of SYS$DISK
    - Default directory string through the Set Default Directory (SYS$SETDDIR) system service

- Base scheduling priority through the Set Priority ($SETPRI) system service
- User name
- User identification code (UIC)

13. After the process's correct UIC has been set, LOGINOUT recreates the job logical name table and possibly the group logical name table.
14. LOGINOUT attempts to change the process name from *_ttcu:* to the user name. This attempt can fail if another process in the same group already has the same name. (A common cause of user name duplication is a user logged in at more than one terminal.) In the case of failure, the process retains its name *(_ttcu:)*, guaranteed to be unique for a given system.
15. LOGINOUT completes the local rights list entries based on the process characteristics and the identifiers associated with the UIC.
16. It stores the job type in the JIB, at offset JIB$L_JOBTYPE.
17. It sets the default message mask (CTL$GB_MSGMASK) to enable display of all message fields and copies the SYSGEN parameter RMS_FILEPROT to establish the process's default file protection (PCB$L_DEFPROT).
18. LOGINOUT copies the remaining attributes extracted from the authorization file to their proper places.

    - It copies information about primary and secondary day restrictions to the JIB.
    - It moves process quotas and limits, testing as appropriate to ensure that they are within system bounds.
    - It stores the authorized priority in PCB$L_AUTHPRI and PHD$L_AUTHPRI.
    - It copies the default privilege mask from the UAF record into CTL$GQ_PROCPRIV and the authorized privileges into PHD$Q_AUTHPRIV.

19. It requests the $SET_RESOURCE_DOMAIN system service to rejoin the system domain with the process's current access rights.
20. It initializes ARB$Q_PRIV and PHD$Q_PRIVMSK as the default privilege mask ORed with the image privilege mask.
21. It builds an object rights block for the terminal based on the process's UIC.
22. It copies the terminal name to PCB$T_TERMINAL.
23. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_CHKRESTRICT routine.
24. LOGINOUT checks a number of other fields in the authorization file record. These include the user or account job limit, the primary and secondary password expiration flags, the DISUSER flag, the account expiration time, and the account hourly restrictions. These checks are waived in the case of the SYSTEM account logging in on the console terminal.

25. If SYS$INPUT is not a remote terminal and reconnection is allowed for the account, LOGINOUT then checks whether the user has disconnected from a process that still exists. It performs a wildcard $GETJPI, looking for a process with the same user name and UIC and a disconnected terminal. It displays any matches and asks the user to which process, if any, the terminal should be connected. It records the answer for later use.
26. If the user does not have OPER privilege, LOGINOUT checks that the interactive process count would not be exceeded by the logging in of this process, and that logins are not currently disabled.
27. If there is a limit to the number of processes that can log in with the user name or account name associated with this process, LOGINOUT scans the PCB vector, counting the number of top-level processes whose JIB user name or account name matches that of this process. It checks that the maximum would not be exceeded by the logging in of this process.
28. LOGINOUT removes SYSPRV from the working privileges unless the user holds it so that files opened subsequently (for example, the CLI, SYS$INPUT, and SYS$OUTPUT) are opened with the user's privileges rather than those of LOGINOUT.
29. LOGINOUT begins initialization for a CLI. It creates user mode logical names PROC1 through PROC9, each equated to the file specification of a command procedure (or indirect command file) to be executed before the CLI enters its input loop. Currently, only PROC1, PROC2, and PROC3 are used. PROC1 is equated to the system name table translation of the logical name SYS$SYLOGIN.

    PROC2 is equated to the file specified by the LGICMD field of the user's UAF record or the file specified by the login qualifier /COMMAND at login time (by an authorized user). If the contents of the LGICMD field are null and no /COMMAND qualifier was present on the login command, PROC2 is equated to the string LOGIN. The LGICMD field should indicate the null device (using the string NL:) to provide a default of no login command file.

    PROC3 is used only for a DECwindows process and is equated to the session manager startup file.

    When the CLI later executes its initialization code, it will translate these logical names and execute the command procedures (or indirect command files).
30. LOGINOUT requests a merged image activation of the selected CLI to map the CLI into the low-address end of P1 space (see Chapter *System Overview*). Except in the case of DCL, LIB$P1_MERGE, in module [VMSLIB]LIBMERGE, first merges the CLI into P0 space to determine its size, deletes the P0 space, and maps the correct amount of P1 space. Next, LOGINOUT requests a merged image activation of the CLI's com-

mand table. Because of the way DCL and its table are built and installed, the image activator can determine their size without having to map them.

Network and DECwindows processes always use DCL and DCLTABLES as the CLI name and command table name. A restricted user receives the CLI name and command table name specified in the UAF record. However, an unrestricted interactive user can specify /CLI and /TABLE on the login command line to choose a particular CLI and command table. If the login command line does not contain a /CLI qualifier, LOGINOUT assigns the first nonzero CLI name in the following list to an unrestricted user:

- CTL$GT_CLI_NAMESTRING, the CLI name specified by the image activator
- CTL$GT_SPAWNCLI, the CLI name specified by a parent process for a spawned subprocess
- The default CLI specified in the UAF record
- CTL$GT_CLINAME, the CLI name of the parent process
- DCL and DCLTABLES

31. LOGINOUT calls a kernel mode procedure to change the owner and protection of the CLI and command table pages. It alters the protection on all pages to allow user mode reads.
32. To accommodate the CLI symbol table, LOGINOUT requests the Expand Process/Control Region ($EXPREG) system service to expand P1 space by a number of pagelets equal to the SYSGEN parameter CLISYMTBL. It updates the global location CTL$GL_CTLBASVA to reflect the new low-address end of P1 space.
33. It opens SYS$INPUT and SYS$OUTPUT.
34. It records the user's account name in CTL$T_ACCOUNT and in the JIB.
35. If the DISWELCOME flag is clear in the UAF record, LOGINOUT writes to SYS$OUTPUT, announcing successful login. It first translates the logical name SYS$WELCOME and writes the welcome message defined by the system manager. If SYS$WELCOME is not defined, LOGINOUT writes the following message, obtaining the version number from the global location SYS$GQ_VERSION and the node name by translating the logical name SYS$NODE:

```
Welcome to OpenVMS Alpha (TM) Operating System, Version V7.0
  on node FOOBAR
```

36. If the DISREPORT flag is clear in the UAF record, LOGINOUT also writes the dates of the last interactive and noninteractive logins and the number of login failures since the last successful login. If the DISNEWMAIL flag is clear, it writes the number of new mail messages for the user.

37. LOGINOUT creates the executive mode logical names SYS$LOGIN, SYS$SCRATCH, and SYS$LOGIN_DEVICE in the process's job logical name table. The equivalence name for each of the first two logical names specifies the default disk and directory specified by the user's UAF record. The equivalence name for the third is simply the default disk. (To override the default disk, follow the user name portion of the login sequence with the qualifier /DISK=ddcu:.)

    For a DECwindows terminal emulation window, LOGINOUT creates the logical name DECW$DISPLAY, with the workstation device name as the equivalence name.

    For a remote login, it creates two logical names representing the remote node's name or address, SYS$REM_NODE and SYS$REM_NODE_FULLNAME. The equivalence name for SYS$REM_NODE is the DECnet Phase IV format of the remote node's name. The equivalence name for SYS$REM_NODE_FULLNAME is either the DECnet Phase V full name of the remote node or the Phase IV node name. LOGINOUT also creates the logical name SYS$REM_ID, whose equivalence name is the remote user name.
38. LOGINOUT checks whether the primary or secondary password lifetime has ended. If so, it marks the password as expired in the UAF record. If the DISFORCE flag is clear in the UAF record or if the user specified the /NEW_PASSWORD qualifier on the login command line, LOGINOUT forces the user to set a new password before continuing. If the DISFORCE flag is set, LOGINOUT informs the user that the password has expired but allows the login to continue.

    If the lifetime of either the primary or secondary password has not ended but is due to expire within five days, LOGINOUT warns the user of that fact.
39. LOGINOUT records the time of login in the UAF record and in CTL$GQ_LASTLOGIN. It updates CTL$GL_LOGFAILS and CTL$GL_LOGIN_FLAGS.
40. If the password was changed, it notifies the security audit subsystem.
41. If this is a user logging into a DECwindows workstation, LOGIN calls a DECwindows routine to start the windows manager.
42. It notifies the security audit subsystem of the login.
43. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_FINISH routine.
44. At this point, LOGINOUT has finished its work and must pass control to the CLI. To pass control to the CLI, LOGINOUT calls an executive mode routine that performs the following actions:

    a. It changes the protection on pages in the PPD region so that the pages can only be accessed from supervisor and inner access modes.
    b. It calls EXE$REI_INIT_STACK, in module REI, passing it the procedure value of the CLI from CTL$AG_CLIMAGE and an access

mode of supervisor. EXE$REI_INIT_STACK reinitializes the executive mode stack to empty, creates an exception stack frame with a saved program counter (PC) and processor status (PS) reflecting its arguments, and executes a `CALL_PAL REI` instruction. This returns the process to supervisor mode with the PC pointing to the first instruction in the CLI, its initialization routine.

### 11.4.2 LOGINOUT and Batch Processes

Many of the operations performed by LOGINOUT for an interactive process are also necessary for a batch process. For example, LOGINOUT must open the input and output streams and map the CLI. However, LOGINOUT does not perform password verification—either the input symbiont has already checked it or, in the case of a SUBMIT command, it is not necessary.

Rather than describing the steps performed by LOGINOUT again, the following list simply specifies those that are different for a batch process:

1. When the batch flag is set in CTL$GL_CREPRC_FLAGS, a copy of the flags originally specified to the $CREPRC system service, LOGINOUT takes actions to create a batch process.
2. It initializes the account name fields in the JIB and P1 space to the string `<batch>`.
3. In routine INIT_BATCH, in module [LOGIN]DETACHED, LOGINOUT requests the $SNDJBCW system service to obtain information about the batch process, for example, its user name, process priority, and working set information.

   The prompted reads of user name and password, and the system announcements that occur in the login of an interactive process, are unnecessary for a batch process.
4. If LOGINOUT callouts are enabled, it initializes callout variables specific to batch processes and calls each image's LGI$ICR_IDENTIFY routine.
5. LOGINOUT establishes the file name to correspond to SYS$INPUT and SYS$COMMAND by concatenating the file IFI (returned by RMS) and the batch input file name. For the file name to correspond to SYS$OUTPUT and SYS$ERROR, it concatenates the file IFI (returned by RMS) with the batch log file name. The files are opened subsequently by code common to batch and interactive processes.
6. LOGINOUT reads the authorization file record for this user. It obtains process attributes to supplement information specified during batch queue creation and job submission. These values from the authorization file are minimized with the values returned by the job controller.
7. The job parameters, P1 through P8, if present, are defined as user mode logical names, which the CLI later translates.
8. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_JOBSTEP routine.

The procedures of mapping the CLI and transferring control are performed by the same code that initializes an interactive process. In both cases, if SYS$SYLOGIN is defined as a system logical name, the first commands that execute are the commands in the site-specific login command file. If the UAF does not specify a user login command file, the command file SYS$LOGIN:LOGIN.COM is executed next.

LOGINOUT regains control at the beginning of each batch job step (see Section 11.8).

### 11.4.3 LOGINOUT and Network Processes

The DECnet ancillary control process (ACP) image is either NETACP for Phase IV DECnet or NET$ACP for Phase V DECnet. The DECnet ACP image requests the SCREPRC system service to create a network process. Many of the operations performed by LOGINOUT for a network process are similar to those for an interactive process.

The DECnet ACP specifies the SCREPRC INPUT, OUTPUT, and ERROR arguments as follows:

- The INPUT argument is the name of a command procedure to be invoked by the CLI that LOGINOUT maps.
- The OUTPUT argument is a set of flags, followed by access control information. The proxy flag indicates whether a proxy login is allowed. The preauthorized flag indicates that the user name in the access control information can be used without a password. NET$ACP uses the preauthorization mechanism to handle default DECnet accounts and user names associated with session control applications and objects.
- The ERROR argument is the address of a network control block (NCB) for the connection.

In routine INIT_NETWORK, in module [LOGIN]DETACHED, LOGINOUT obtains the network logical link number from the NCB and stores the remote node name, address, and ID in P1 space. It checks to see whether the network process should use proxy login or the access control information. In the latter case, it reads the system authorization file and, if LOGINOUT callouts are enabled, calls each image's LGI$ICR_AUTHENTICATE and LGI$ICR_IDENTIFY routines. It creates an executive mode logical name SYS$NET, whose equivalence name is the address of the NCB.

LOGINOUT activates DCL to execute the command procedure and creates a log file. In most cases, the command procedure for DECnet Phase IV is NETSERVER.COM and for DECnet Phase V, NET$SERVER.COM.

## 11.5 CLIS AND IMAGE PROCESSING

Digital provides for the possibility of multiple CLIs, although only DCL is supplied with the OpenVMS software. This section describes various features of DCL, in particular, those that are typical of any CLI.

After the DCL CLI gains control and performs some initialization, it reads and processes successive records from SYS$INPUT. This section describes those operations that result in image execution, to contrast interactive and batch processes with processes that do not map a CLI.

One important step performed by the DCL CLI is to request the Declare Exit Handler ($DCLEXH) system service to declare the routine DCL$EXIT-HAND in module [DCL]IMAGEXECT, as a supervisor mode exit handler. It is this handler that prevents process deletion following image exit and allows the successive execution of multiple images within the same process. Section 11.7 summarizes the flow of control within a process that runs multiple images.

### 11.5.1 CLI Initialization

The DCL CLI's initialization code is the routine DCL$STARTUP in module [DCL]INITIAL. Running in supervisor mode, the initialization code performs the following steps before entering the main command processing loop:

1. Entered in supervisor mode, DCL resets the supervisor mode stack pointer (SP) to its base and then calls DCL$STARTUP, creating an initial procedure frame on the supervisor mode stack. This initial procedure frame contains zero in its saved frame pointer (FP), terminating the call chain. DCL calls itself again and establishes a procedure frame condition handler.
2. DCL writes the address of its CLI callback service routine in the global location CTL$AL_CLICALBK. Callback is a mechanism an image uses to obtain services from the CLI, such as symbol creation and lookup.
3. DCL initializes its work area from internal variables transferred by LOGINOUT to the PPD region, pointed to by CTL$AG_CLIDATA. It initializes the CLI symbol table data structures and defines the reserved symbols $STATUS, $SEVERITY, and $RESTART.
4. For a batch process, DCL translates the logical names for BATCH$RESTART and parameters P1 through P8. It creates symbols whose values are the equivalence names.
5. It initializes various RMS data structures to describe the process-permanent files SYS$INPUT and SYS$OUTPUT.
6. It examines CTL$GL_UAF_FLAGS and notes whether the account is captive or restricted.
7. It defines the logical name SYS$OUTPUT.
8. DCL translates PROC1 through PROC9 and saves their equivalence names to identify the command procedures it must execute.
9. It calls SYS$RMSRUNDWN to close the files opened by LOGINOUT.
10. DCL requests the $RUNDWN system service with an argument of user mode to run down the LOGINOUT image.

11. DCL validates the structure of its command table.
12. It issues a special I/O request to the terminal driver, naming the current process as the terminal owner.
13. DCL enables CTRL/Y and out-of-band ASTs on the terminal. (CTRL/Y ASTs are not enabled if the UAF record had the DISCTLY flag set.)
14. DCL requests the Declare Change Mode Handler (SDCLCMH) system service to establish a change-mode-to-supervisor handler. This handler allows the CLI to enter supervisor mode from user mode when it needs to access write-protected data structures. One instance where this is required is in symbol definition, because CLI symbol tables are protected from write access by user mode.
15. Finally, DCL branches to the first instruction of the main command processing loop, routine DCL$RESTART in module [DCL]COMMAND, an alternative entry point for DCL$CMDSTART, in module [DCL]COMMAND.

### 11.5.2 Command Processing Loop

In its main command processing loop, a CLI reads a record from SYS$INPUT and takes whatever action is dictated by the command. DCL can perform some actions directly. Others require the execution of a separate image. Table 11.3 lists the general operations performed by DCL and indicates those actions that require an external image.

A simplified flow of control through the DCL CLI is pictured in Figure 11.5. Except for the two boxes labeled otherwise, all boxes in Figure 11.5 represent DCL CLI actions.

After a CLI reads a record from the input stream and recognizes a command, it either performs the requested action itself or activates an external image. DCL can execute some commands without destroying a currently executing image (see Table 11.4). Any other command either requires an image

Table 11.3 General Actions Performed by the DCL CLI

| *General CLI Operations* | *Sample Commands* |
|---|---|
| Commands that the CLI can execute internally (see Table 11.4) | EXAMINE, SET DEFAULT |
| Commands that require external images | COPY, LINK, some SET commands, some SHOW commands |
| Commands that require internal processing and an external image | LOGOUT, MCR, RUN, EOJ |
| Foreign command definition | *command-string* :== *$image-file-spec* |
| Automatic foreign command | *image-file-name, command-procedure-file-name* |
| Other operations that destroy an image | STOP, EXIT |
| Invoking a command procedure | *command-file-spec* |
| Other CLI operations | Symbol definition, expression evaluation |

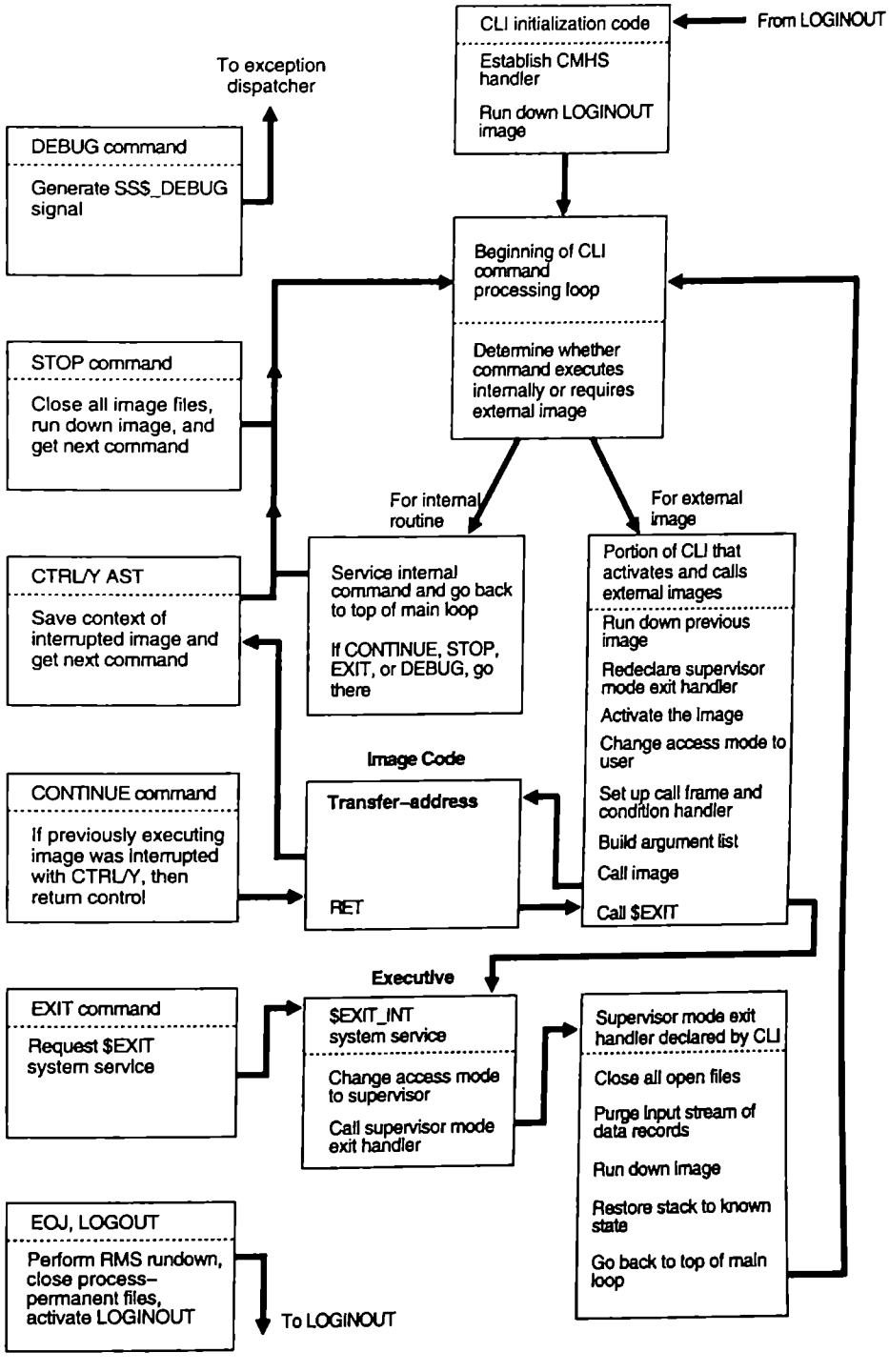

CLI initialization code
From LOGINOUT
Establish CMHS handler
Run down LOGINOUT image
To exception dispatcher
DEBUG command
Generate SS$_DEBUG signal
Beginning of CLI command processing loop
Determine whether command executes internally or requires external image
STOP command
Close all image files, run down image, and get next command
For internal routine
For external image
CTRL/Y AST
Save context of interrupted image and get next command
Service internal command and go back to top of main loop
If CONTINUE, STOP, EXIT, or DEBUG, go there
Portion of CLI that activates and calls external images
Run down previous image
Redeclare supervisor mode exit handler
Activate the image
Change access mode to user
Set up call frame and condition handler
Build argument list
Call image
Call $EXIT
Image Code
Transfer-address
RET
CONTINUE command
If previously executing image was interrupted with CTRL/Y, then return control
Executive
EXIT command
Request $EXIT system service
$EXIT_INT system service
Change access mode to supervisor
Call supervisor mode exit handler
Supervisor mode exit handler declared by CLI
Close all open files
Purge input stream of data records
Run down image
Restore stack to known state
Go back to top of main loop
EOJ, LOGOUT
Perform RMS rundown, close process-permanent files, activate LOGINOUT
To LOGINOUT

**Figure 11.5**
Simplified Control Flow Through the DCL CLI

Table 11.4 Commands Handled by CLI Internal Procedures

| *Command* | *Description* |
|---|---|
| @ | Execute a command procedure |
| - | Create/modify a symbol |
| ALLOCATE | Allocate a device |
| ASSIGN | Create a logical name |
| ATTACH | Transfer control to another process in job |
| CALL | Transfer control to a labeled subroutine in a command procedure |
| CANCEL | Cancel scheduled wakeups for a process |
| CLOSE | Close a process-permanent file |
| CONNECT | Connect the physical terminal to a virtual terminal of another process |
| CONTINUE | Resume interrupted image |
| CREATE/NAME_TABLE | Create a new logical name table |
| DEALLOCATE | Deallocate a device |
| DEASSIGN | Delete a logical name |
| DEBUG | Invoke the symbolic debugger |
| DECK | Delimit the beginning of an input stream |
| DEFINE | Create a logical name |
| DEFINE/KEY | Associate a character string and attributes with a terminal key |
| DELETE/KEY | Delete a key definition |
| DELETE/SYMBOL | Delete a symbol definition |
| DEPOSIT | Modify a memory location |
| DISCONNECT | Disconnect a physical terminal from a virtual terminal |
| ENDSUBROUTINE | Label the end of a called subroutine in a command procedure |
| EOD | Delimit the end of an input stream |
| EXAMINE | Examine a memory location |
| EXIT | Exit a command procedure<br>Run down an image after invoking exit handlers |
| GOSUB | Transfer control to a labeled subroutine in a command procedure |
| GOTO | Transfer control within a command procedure |
| IF/THEN/ELSE/ENDIF | Conditional command execution |
| INQUIRE | Interactively assign a value to a symbol |
| ON | Define conditional action |
| OPEN | Open a process-permanent file |
| READ | Read a record into a symbol |
| RECALL | Display previously entered commands for possible reissue |
| RETURN | Terminate a GOSUB subroutine procedure |
| SET CONTROL | Determine responses to CTRL/C, CTRL/Y, and CTRL/T |
| SET DEFAULT | Define default directory string |
| SET KEY | Change current terminal key definition state |
| SET [NO]ON | Determine error processing |
| SET OUTPUT_RATE | Set rate at which output is written to a batch job log file |
| SET PREFIX | Set a prefix control string for verified command lines |
| SET PROMPT | Change the CLI's prompt string |
| SET PROTECTION | Define default file protection |
| SET SYMBOL | Alter scope of a symbol |
| SET UIC | Change process UIC and default directory string |
| SET [NO]VERIFY | Determine echoing of command procedure commands |
| SHOW DEFAULT | Display default directory string |
| SHOW KEY | Display terminal key definitions |

Table 11.4 *(continued)*

| *Command* | *Description* |
|---|---|
| SHOW PROTECTION | Display default file protection |
| SHOW QUOTA | Display current disk file usage |
| SHOW STATUS | Display status of currently executing image |
| SHOW SYMBOL | Display value of symbol(s) |
| SHOW TIME | Display current time |
| SHOW TRANSLATION | Show translation of single logical name |
| SPAWN | Create a subprocess and transfer control to it |
| SUBROUTINE | Label beginning of called subroutine in a command procedure |
| STOP | Run down an image, bypassing exit handlers |
| WAIT | Wait for specified interval to elapse |
| WRITE | Write the value of a symbol to a file |

to execute (such as COPY or LINK) or directly affects the currently executing image (such as STOP).

### 11.5.3 Image Initiation by a CLI

When a CLI determines that an external image is required, it first performs some command-specific steps. It then enters a common routine to activate and call the image. DCL's common routine is DCL$EXTIMAGE, in module [DCL]IMAGEXECT. Many of the steps it takes are nearly identical to the steps performed by EXE$PROCSTRT, described in Chapter 7:

1. If an image is active, DCL enters user mode and requests the $EXIT system service, which first calls any user mode exit handlers and then supervisor mode exit handlers. Control returns to DCL's previously declared supervisor mode exit handler. DCL calls SYS$RMSRUNDWN to close any files opened by the image. It requests the $RUNDWN system service with an argument of user mode to remove any traces of the previously executing image. If the previous image terminated normally, these actions are unnecessary. However, if the user typed CTRL/Y followed by an external command, the normal image termination path is bypassed; DCL must perform this extra step to ensure that the previous image is eliminated before another is activated.
2. DCL declares DCL$EXITHAND as a supervisor mode exit handler to regain control when the image exits. Recall from Chapter 8 that an exit handler must be redeclared after each use.
3. To activate the image, DCL requests the $IMGACT system service, described in Chapter 8.
4. If the activation succeeds, DCL enters user mode and calls a DCL procedure to create an initial procedure frame on the user mode stack. This initial procedure frame contains zero in its saved FP, terminating the call chain.

5. DCL establishes EXE$CATCH_ALL, in module PROCSTRT, as the procedure frame condition handler and as the last chance exception handler.
6. It requests the $IMGFIX system service to perform address relocations and fixups.
7. DCL builds an argument list to pass to the image and to any intervening procedures such as SYS$IMGSTA. The list consists of the following arguments:

   a. Address of image transfer address array
   b. Address of CLI utility routine dispatcher
   c. Address of image header
   d. Address of image file descriptor
   e. Link flags taken from the image header
   f. CLI flags

8. DCL calls the image at the first address in the transfer address array, described in Chapter 8. Unless the image was linked with the /NOTRACEBACK qualifier, the first transfer address entry is the address of the $IMGSTA system service. This service establishes the traceback exception handler and maps the debugger, if requested.
9. Later, the image terminates itself by returning to its caller or by requesting the $EXIT system service. Since DCL requests the $EXIT system service anyway, the termination method chosen by the image is generally irrelevant. However, for an image that might be called as a procedure from another image, returning is the preferred method of image termination.

### 11.5.4 Normal Image Termination

When an image in a process with a CLI terminates normally, the $EXIT system service requests the $EXIT_INT system service, which eventually calls the supervisor mode exit handler established by the CLI before it called the image. DCL's exit handler, DCL$EXITHAND in module [DCL]IMAGEXECT, performs several cleanup steps:

1. If the image exited with an error status in R0, the handler writes the corresponding error message.
2. It calls SYS$RMSRUNDWN, closing any files left open by the image and the image file itself, leaving process-permanent files open.
3. It discards any data records in the input stream (records that do not begin with a dollar sign for DCL) and issues a warning message.
4. It runs down the terminated image by requesting the $RUNDWN system service with an argument of user mode.
5. It stores final image status and severity in the symbols $STATUS and $SEVERITY. If an error or severe error occurred and the process had entered the DCL command ON ERROR or ON SEVERE_ERROR, DCL

executes the rest of that command line. Otherwise, it continues with the next step.

6. It transfers control to the beginning of the main command loop so that the CLI can read and process the next command.

### 11.5.5 Abnormal Image Termination

A user can interrupt an image by typing CTRL/Y or CTRL/C; an image can interrupt itself through the pause capability supplied by the Run-Time Library procedure LIB$PAUSE. Further execution of the image depends on the sequence of commands issued while the image is interrupted.

**11.5.5.1 CTRL/Y Processing.** When CTRL/Y is typed at the terminal, the terminal driver transfers control to the supervisor mode AST procedure established by DCL during its initialization. The AST procedure first reestablishes itself, enabling future CTRL/Ys to be passed to the same AST procedure. It then checks whether the process has disabled CTRL/Ys through the SET NOCONTROL=Y command. If so, the AST procedure returns, dismissing the CTRL/Y. Otherwise, its actions depend on the access mode interrupted by the CTRL/Y.

If the previous mode was supervisor, the AST procedure actions depend on whether an ON CONTROL_Y command was issued previously, specifying a particular command to be executed in response. If so, the AST procedure sets a flag to request that the command be executed and returns. If not, DCL is restored to its initial state (with no nesting of indirect levels) and control transfers to the beginning of the main command loop.

If the previous mode was user, the CTRL/Y interrupted an image. DCL requests the $SINGLE_THREAD system service to place any other kernel threads of this process into a quiescent state. It requests the Clear AST ($CLRAST) system service to enable subsequent supervisor mode ASTs to be delivered to the current kernel thread. If the process had entered the DCL command ON CONTROL_Y, DCL executes the rest of that command line. Otherwise, if the image was installed with enhanced privileges, DCL saves those privileges and resets the process privileges to those in use before the image was activated. After setting a flag, DCL returns to command processing. If, at this point, the user enters the DCL commands ATTACH, CONTINUE, or SPAWN, the appropriate action is taken and the image is not run down.

Any other command causes DCL to run down a privileged image before executing the command; a nonprivileged image may continue (see Section 11.5.5.3). Issuing a STOP command for a nonprivileged image causes DCL to terminate the image without calling user mode exit handlers (see Section 11.5.5.7). However, because a privileged image is run down before the STOP command is processed, its exit handlers are called.

11.5.5.2 **Pause Capability.** The Run-Time Library procedure LIB$PAUSE provides the capability to interrupt an image under program control. An image executing in the context of an interactive process can call LIB$PAUSE to interrupt itself and transfer control to the CLI at the beginning of its main command loop.

11.5.5.3 **State of Interrupted Images.** When a nonprivileged image is interrupted, the image context is saved and control is transferred to the beginning of DCL's main command loop, allowing the user to execute commands. If the command is one that DCL can perform internally (see Table 11.4), the image context is not destroyed and the image can be continued.

However, execution of any command that requires an external image destroys the context of the interrupted image. In addition, executing an indirect command file destroys an interrupted image, even if the commands in the indirect command file can be performed internally by DCL.

Six commands that the user can enter when an image has been interrupted by CTRL/Y have special importance. These commands are ATTACH, CONTINUE, DEBUG, EXIT, SPAWN, and STOP. ATTACH and SPAWN are described in Section 11.6. The other commands are described in the following sections.

11.5.5.4 **CONTINUE Command.** If a CONTINUE command is typed and the previous mode was user, DCL tests whether the image being executed is one installed with enhanced privileges, for example, SHOW.EXE, the SHOW utility. If so, it restores those privileges. In either case, it requests the $SINGLE_THREAD system service to awaken the other kernel threads, dismisses the AST, and returns control to the image at the point where it was interrupted.

11.5.5.5 **DEBUG Command.** If a DEBUG command is typed and the previous mode was user, DCL generates a user mode SS$_DEBUG signal, as described in Chapter 8. The signal is handled by the condition handler established by the $IMGACT system service. (If the image was linked with the /NOTRACEBACK qualifier, the handler was never established and the image exits.) This handler responds to the SS$_DEBUG signal by mapping the debugger (if it is not already mapped) and transferring control to it. This technique enables the debugger to be used even if the image was not linked with the /DEBUG qualifier.

11.5.5.6 **EXIT Command.** In response to an EXIT command entered by an interactive user, DCL disables CTRL/Y ASTs and tests whether a CTRL/Y or CTRL/C was just entered. If so, it tests whether the previous mode was user. If it was, DCL restores any enhanced privileges the image had. It enters user mode and requests the Force Exit ($FORCEX) system service to exit itself (see

Chapter 6), which queues a user mode AST to the current kernel thread. The AST procedure requests the $EXIT system service, which results in calling exit handlers and running down the image.

11.5.5.7 **STOP Command.** In response to a STOP command, DCL first determines whether an image or a process is being stopped. (The various STOP commands are described in the *OpenVMS DCL Dictionary*.)

If an image is being stopped, it tests whether the process is interactive. If so, DCL closes all files opened by the image and the image file or files, by calling SYS$RMSRUNDWN. It cancels DCL$EXITHAND, its own exit handler. It then requests the $RUNDWN system service with an argument of user mode. Finally, it transfers control to the beginning of the main command loop.

Note that STOP performs nearly identical operations to the exit handler called as a result of an $EXIT system service request or an EXIT command. The only difference between the EXIT sequences and the STOP command is that exit handlers are not called when an image terminates with a CTRL/Y STOP sequence. Thus, in most cases, the STOP and EXIT commands are interchangeable. One useful aspect of the STOP command is that it can eliminate an image containing a user mode exit handler that is preventing completion of image deletion either intentionally or as the result of an error.

## 11.6 SPAWN AND ATTACH

DCL provides two commands to create and connect with interactive subprocesses. The DCL command SPAWN creates interactive subprocesses. The ATTACH command transfers terminal control from one process to another within the same job. The module [DCL]SPAWN contains the code for both commands. The Run-Time Library procedures LIB$SPAWN and LIB$ATTACH make the SPAWN and ATTACH functions available to an image by passing the request back to the DCL CLI. The major difference between the two ways of requesting the functions is the method of passing parameters. From DCL level, the command line is parsed to obtain the parameters. The Run-Time Library procedures use an argument list.

### 11.6.1 SPAWN

Spawning a subprocess primarily involves copying process context information from the creating process to the subprocess. This information includes the process CLI symbols, process-private logical names, current privileges, out-of-band AST settings, verify flag settings, prompt string, default disk and directory, keypad definitions and states, and the command line that was passed to SPAWN (if one exists).

In response to a SPAWN request, the DCL routine DCL$SPAWN, in module [DCL]SPAWN, running in supervisor mode, performs the following operations:

1. It parses the command line to determine what qualifiers are present. It validates the qualifiers and copies them to a temporary data structure.
2. It temporarily disables the current process's out-of-band ASTs, blocking CTRL/Y ASTs during a critical section of code.
3. If the spawn request originated in the LIB$SPAWN module and the caller asked that an event flag be set at completion, DCL clears the specified event flag.
4. It creates or locates a termination mailbox and requests an attention AST if a message is written to the mailbox.

   Termination information from the subprocess is written to the termination mailbox when the subprocess is eventually deleted. The attention AST is delivered to the subprocess's creator at that time. Because a termination mailbox is created with enough quota to handle termination messages from four spawned subprocesses, DCL checks for an available one that the new subprocess can share before creating a new mailbox.
5. DCL records the name of the subprocess's CLI and command table files in P1 space locations CTL$GT_SPAWNCLI and CTL$GT_SPAWNTABLE. The $CREPRC system service later copies them to the process quota block (PQB). When LOGINOUT eventually runs in the context of the new subprocess, this is the CLI that it will invoke. The default, if no CLI or no command table is specified, is the creator's CLI or command tables.
6. For CLIs supplied by Digital, DCL creates a second mailbox, called the communication mailbox, through which further context information is transferred to the spawned subprocess, as described in step 12.
7. DCL creates an attach request mailbox for the current process with a jobwide logical name of the form DCL$ATTACH_*pid*, where *pid* is the extended process ID. Other processes in the job tree can attach to this process by writing attach requests to this mailbox.

   If the spawn request does not specify an explicit SYS$OUTPUT device for the subprocess's output, DCL creates an output mailbox to funnel subprocess writes to the current process's SYS$OUTPUT. It enables a write attention AST on the output mailbox.
8. DCL requests the $GETJPI system service to determine the current process's nondeductible quotas, base priority, and current image count. From these quotas, it builds a quota list to be used in the creation of the spawned subprocess.
9. If the process name was not specified in the command line or argument list, DCL creates one by appending _*n* to the user name string, where *n*

is a value from 1 to 255. If the new name is a duplicate, DCL increments *n* and tries again.

10. If the command line or argument list specified that the subprocess should be created with the authorized privileges of its creator, DCL requests the Set Privileges ($SETPRV) system service to reset the working privileges of the current process to those that were authorized. (If the process is running an image installed with privilege, its working privileges could have been increased temporarily.)

    DCL requests the $CREPRC system service to create the subprocess. It specifies LOGINOUT as the IMAGE argument and the name of the communication mailbox from step 6 as the ERROR argument. If the creating process does not specify input and output files to the SPAWN command, DCL uses the creating process's SYS$INPUT and SYS$OUTPUT file specifications as the INPUT and OUTPUT arguments. It specifies the termination mailbox from step 4 to the $CREPRC service to receive a process deletion message from the subprocess. Because the $CREPRC request does not include a privilege mask for the subprocess, the $CREPRC system service creates the subprocess with whatever the current working privileges are (see Chapter 7).

    If DCL requested $SETPRV to reset the privileges to those that were authorized, it requests $SETPRV again to restore the working privileges that were in effect when the SPAWN was requested.

11. When LOGINOUT runs in the context of the newly spawned subprocess, it maps the specified CLI, DCL in this example, and passes control to it. DCL determines that it is running in the context of a subprocess and translates the logical name SYS$ERROR. If there is a supervisor mode translation with a mailbox name as the equivalence string, DCL recognizes that a SPAWN operation is in progress and that it must read context information from the creating process.

    At this point, both the creating process and the spawned subprocess are executing DCL routines. The creating process passes context information to the spawned subprocess in the following manner:

    a. The spawned subprocess assigns a channel to the communication mailbox and issues read requests to it.
    b. The creating process requests the Queue I/O Request and Wait ($QIOW) system service to write context information to the mailbox, one record at a time. Because the request has the IO$M_NOW modifier, the request can complete without a matching read. Each record has a type code identifying its contents. When the subprocess receives the information, it adds the information to its context.
    c. The first transferred record contains the permanently enabled privilege mask (CTL$GQ_PROCPRIV), verify and other flag settings, out-of-band AST mask, and prompt string.

The spawned subprocess reads the record and initializes the process accordingly. It requests the $SETPRV system service to disable all privileges, then resets the process privileges from those transferred in the record. Thus, the working, permanently enabled (current), and authorized privilege masks of the subprocess contain the privileges its creator possessed when the spawn occurred, unless the spawn requested that the subprocess be created with only the authorized privileges. This enables a privileged image to tailor the environment in a spawned subprocess.

d. Next, the creating process transfers the SPAWN command string (if one was specified).
e. Unless the SPAWN command specified that logical names should not be copied, the creating process scans the process logical name directory, which contains a list of process logical name table names. It copies all table names that were defined in user or supervisor mode and that do not have the CONFINE attribute. It copies all the logical names defined in those tables. The spawned subprocess creates the corresponding logical name tables and their logical names.
f. Unless the SPAWN command specified that symbols should not be copied, the creating process transfers the contents of the symbol table, one symbol at a time. The spawned subprocess receives each symbol and places it into the subprocess's symbol table.
g. Unless the SPAWN command specified that keypad definitions should not be copied, the creating process transfers terminal keypad definitions.

Note that the creating process's potentially modified DCL command tables are not transferred to the subprocess.

12. Once it has transferred all information to the subprocess, the creating process writes an end-of-file message to the communication mailbox without the IO$M_NOW modifier. The I/O does not complete until the subprocess has read all the preceding messages and the end-of-file message. The creating process tests whether it should wait for the subprocess. If so, it requests a write attention AST on the attach request mailbox and hibernates. Otherwise it restores out-of-band ASTs and resumes normal processing.
13. The spawned subprocess deassigns the channel to the communication mailbox. It deletes the supervisor mode logical name SYS$ERROR, leaving the executive mode logical name. It restores out-of-band ASTs and, if the subprocess is interactive, issues a special I/O request to the terminal driver to declare the subprocess the terminal owner. It then continues normal DCL processing.

When a subprocess created by the SPAWN command is deleted, a termination message is written to its creator's termination mailbox. As a result, a write attention AST is queued to the creator. The AST procedure reads the termination message and performs cleanup work pertaining to the deleted subprocess. It deassigns the channels to the communication and output mailboxes, causing them to be deleted. It decrements the count of processes using the termination mailbox and, if the count goes to zero, deassigns the channel, deleting the mailbox. If the subprocess was created by a call to LIB$SPAWN and if an event flag or AST procedure was specified in the call, and the creating process's image count is the same as when the subprocess was spawned, the event flag is set or the AST is delivered.

### 11.6.2 ATTACH

The DCL ATTACH request transfers terminal control from the process that issues the command to a target process. The operation of the DCL ATTACH routine, DCL$ATTACH in module [DCL]SPAWN, running in supervisor mode, is as follows:

1. From the context of the issuing process, DCL first obtains the name or process identification (PID) of the target process. It disables out-of-band ASTs, blocking delivery of CTRL/Y ASTs. It then verifies that the target process is not itself and that it is a process in the same job tree.
2. DCL creates an attach request mailbox for the issuing process and a logical name for the mailbox. Since interactive input will be detached from the issuing process and attached to the target process, the issuing process must have an attach request mailbox to accept attach requests later. Otherwise, the terminal cannot be reattached to it.
3. DCL assigns a channel to the target process's attach request mailbox, whose logical name is DCL$ATTACH_*pid*. DCL writes the name of the current input stream (usually the equivalence name of SYS$INPUT) to the mailbox. Since the target process had declared a write attention AST on its attach request mailbox, it is notified of the message placed in the mailbox. The original process then issues a read request on the target process's attach request mailbox in anticipation of a message from the target.
4. The target process wakes in response to the write attention AST. The AST procedure determines whether the target process is already attached to a terminal. If not, it writes an affirmative response (a longword with a value of 1) to the attach request mailbox. Otherwise, it writes a zero longword to refuse the attach request and reenables the write attention AST for the attach request mailbox.
5. Once it receives the affirmation, DCL in the issuing process deassigns its channel to the target process's attach request mailbox. It requests a write

attention AST for its own attach request mailbox so it can be notified of any incoming attach requests. It then hibernates.

6. The AST procedure in the target process issues a wake request to return control to the target process.

## 11.7 MULTIPLE-IMAGE PROCESSES

A process is created to execute multiple images when code in another process requests the $CREPRC system service specifying LOGINOUT as the image to be run. LOGINOUT maps a CLI into the new process. Its presence enables the execution of multiple images. Figure 11.6 summarizes the flow of control in such a process. Various steps in it have been described in previous sections of this chapter.

When the process is created, LOGINOUT is activated and control is passed to it as in the steps shown in Figure 11.1. The numbers in the following list correspond to the numbers in Figure 11.6, which begins with execution within LOGINOUT.

① Running in user mode, LOGINOUT requests the Change to Executive Mode ($CMEXEC) system service.

② In executive mode, it requests the activation of the DCL CLI and its command tables.

③ LOGINOUT executes a `CALL_PAL REI` instruction to leave executive mode and enter DCL in supervisor mode.

④ Running in supervisor mode, DCL requests the $RUNDWN system service to remove all traces of LOGINOUT and delete the virtual address space it occupied. DCL prompts, if this is an interactive process, and reads a command from SYS$INPUT. It parses the command line to determine which image to activate.

⑤ DCL requests the $DCLEXH system service to declare DCL$EXITHAND as a supervisor mode exit handler.

⑥ It requests the $IMGACT system service to activate the requested image and executes a `CALL_PAL REI` instruction to leave supervisor mode and enter user mode.

⑦ Running in user mode, the DCL CLI establishes EXE$CATCH_ALL as the last chance condition hander (see Chapter *Condition Handling*) for user mode. It requests the $IMGFIX system service to perform relocations and fixups for the images activated in step 6. It calls the image's first transfer address (see Chapter 8), which is typically the $IMGSTA system service.

⑧ EXE$IMGSTA, in module SYSIMGSTA (see Chapter 8), optionally activates the debugger, based on image link and $CREPRC flags.

⑨ If it does not activate the debugger, it establishes a condition handler for the current procedure frame.

⑩ It calls the image's next transfer address. Each transfer address except the last, which is the entry point for the main image, represents a procedure

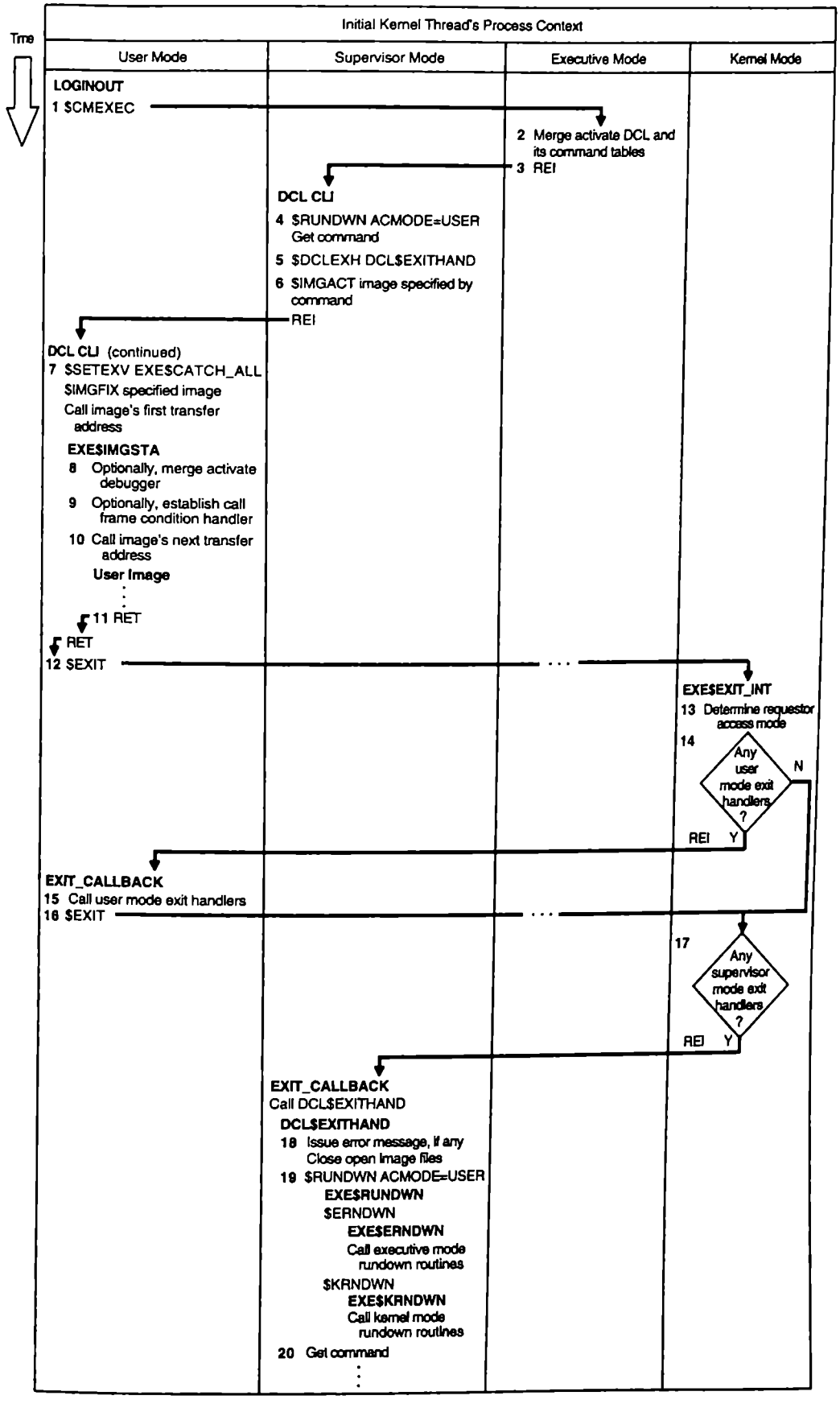



**Figure 11.6**
Process That Executes Multiple Images

that performs some initialization and then calls the next transfer address. Eventually the main image is entered and begins execution.

⑪ When the main image finishes its task, it exits. In this example, it exits by returning to its caller, DCL. Still in user mode, DCL calls EXE$EXIT_IMAGE, in module PROCSTRT. EXE$EXIT_IMAGE requests the $EXIT system service.

⑫ The $EXIT system service, a mode of caller service, requests the $EXIT_INT system service.

⑬ EXE$EXIT_INT, in module SYSEXIT, determines the access mode from which it was requested.

⑭ It tests whether any exit handlers have been declared for the requestor's mode, in this case, user mode. In this example, the user image declared at least one exit handler. EXE$EXIT_INT cleans up the kernel stack and executes a `CALL_PAL REI` instruction to return to user mode at local routine EXIT_CALLBACK.

⑮ EXIT_CALLBACK calls each user mode exit handler declared by the image and its shareable images.

⑯ To return to kernel mode, EXIT_CALLBACK requests the $EXIT system service. (The control flow in the figure is slightly simplified in that it omits EXE$EXIT's request of $EXIT_INT and reentry into EXE$EXIT_INT.)

⑰ EXE$EXIT_INT next checks whether any supervisor mode exit handlers have been declared. DCL has declared one. EXE$EXIT_INT executes a `CALL_PAL REI` instruction to return to supervisor mode.

Running in supervisor mode, EXIT_CALLBACK calls DCL$EXITHAND.

⑱ DCL$EXITHAND outputs any appropriate error message and closes open image files.

⑲ It requests the $RUNDWN system service with an access mode argument of user to remove all traces of the image that just exited. $RUNDWN requests $ERNDWN, which calls any executive mode rundown routines. $RUNDWN requests $KRNDWN, which calls any kernel mode rundown routines and resets the APLD vector (see Chapter *System Service Dispatching*) to clear all image-temporary rundown routines.

⑳ DCL gets the next command and repeats its processing at steps 5, 6, and so on.

## 11.8 LOGINOUT'S OTHER ROLES

LOGINOUT, the image that performs the initialization of an interactive or batch process, also executes to delete such a process. When LOGINOUT executes, it performs login, logout, or batch job step initialization. (When a batch process is submitted with more than one command procedure specified, each procedure is handled as a separate batch job step.) LOGINOUT

determines whether the process is logged in already by the existence of the PPD region, used to communicate between LOGINOUT and the CLI.

If the PPD region exists, LOGINOUT's actions depend on whether the process is interactive or batch. For an interactive process, LOGINOUT, in routine LOGOUT in module [LOGIN]LOGIN, performs the following steps:

1. It enters supervisor mode to cancel any supervisor mode exit handlers declared by the CLI. As a result, if LOGINOUT aborts or is forcibly exited, the process will be deleted anyway. From supervisor mode, it requests the Cancel I/O on Channel ($CANCEL) system service to cancel any outstanding I/O requests on SYS$INPUT, in particular requests for attention or out-of-band ASTs.
2. It enters executive mode to change the protection on the PPD region so that it can write to it from user mode.
3. LOGINOUT copies the IFIs for SYS$INPUT and SYS$OUTPUT from PPD locations into RMS data structures. This restores definitions of SYS$INPUT and SYS$OUTPUT made at login.
4. LOGINOUT notifies the security audit subsystem of the logout.
5. If the user specified the /[NO]HANGUP qualifier on the LOGOUT command, LOGINOUT checks whether it is appropriate to change the terminal characteristics. If the process is interactive and not a subprocess, and the terminal is local, LOGINOUT reads the current terminal characteristics and resets them, altering the hangup bit.
6. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_LOGOUT routine.
7. LOGINOUT writes the logout message to the restored SYS$OUTPUT. (Thus, it cannot be redirected via a logical name definition.) If the user asked for a full logout message, LOGINOUT requests the $GETJPI system service to get information, such as CPU time, number of page faults, and number of I/O requests.
8. It closes SYS$INPUT and SYS$OUTPUT.
9. Finally, LOGINOUT requests the $EXIT system service from executive mode. As described in Chapter 8, this limits the search for exit handlers to the executive mode list, bypassing the supervisor mode exit handler established by the CLI to prevent process deletion following image exit.
10. After any executive mode exit handlers have performed their work, the $EXIT system service requests the $DELPRC system service, which removes the logged out process from the system.

If the process is a batch process, LOGINOUT, in routine INIT_BATCH in module [LOGIN]DETACHED, first closes SYS$INPUT. It requests the $SNDJBCW system service again to determine if there is another job step. If the batch process was submitted with multiple command procedures specified, LOGINOUT opens the new SYS$INPUT and reinitializes the batch

process environment. If LOGINOUT callouts are enabled, it calls each image's LGI$ICR_JOBSTEP routine. It reenters the CLI.

If the previous batch job step failed, or the message that is returned from the job controller indicates that the process should be terminated, LOGINOUT terminates it as follows:

1. It writes a logout message to the log file.
2. It closes the log file.
3. If the log file is to be printed, LOGINOUT requests the $SNDJBCW system service again, this time to queue the file to a print queue.
4. LOGINOUT then requests the $EXIT system service from executive mode. After any executive mode exit handlers have performed their work, the $EXIT system service requests the $DELPRC system service, which removes the process from the system.

## 11.9 RELEVANT SOURCE MODULES

Source modules described in this chapter include

[CLIUTL]SUBMIT.B32
[DCL]COMMAND.MAR
[DCL]EXIT.MAR
[DCL]IMAGECTRL.MAR
[DCL]IMAGEXECT.MAR
[DCL]INITIAL.MAR
[DCL]SPAWN.MAR
[JOBCTL]UNSOLICIT.B32
[JOBCTL]UTILITY.B32
[LOGIN]DETACHED.B32
[LOGIN]INITUSER.B32
[LOGIN]INTERACT.B32
[LOGIN]LOGIN.B32
[TTDRVR]TTYCHARI.MAR

# A Lock and Resource Use by OpenVMS Components

Many OpenVMS facilities use lock management system services to coordinate their own activities, both locally and within a VMScluster system. This appendix examines a number of those facilities and describes their lock use. The aim is to demonstrate a variety of locking techniques and to provide examples of situations where specific techniques are beneficial.

This appendix is by no means a complete description of OpenVMS lock use or of the various facilities mentioned. It assumes that the reader is familiar with Chapter 4 of this book and with the description of the OpenVMS lock management system services found in the *OpenVMS System Services Reference Manual.*

## A.1 ASPECTS OF RESOURCE AND LOCK USE

The data structure that represents the entity being locked is a resource block, commonly referred to as a resource. A resource is uniquely identified by the combination of its resource name string, resource domain, access mode, and parent resource, if any.

A lock on a resource is characterized by its lock mode, the extent to which it allows shared access with other locks on the same resource. Chapter 4 lists the different lock modes: concurrent read/write (CR, CW), protected read/write (PR, PW), null (NL), and exclusive (EX). It describes the compatibility of the various modes and the conditions under which locks are granted, converted, or made to wait for blocking locks to be released.

The resource name string of a resource created by OpenVMS for its own use typically begins with a facility code. The remainder of the string further identifies the specific resource, for example, SCSNODE, device name, or file ID.

Table A.1 lists some OpenVMS facilities, their associated facility codes, and the sections in this appendix that further describe the facility's lock use.

The resource domain defines the name space applicable to a resource and its locks; a resource name is only meaningful within a resource domain. The domain determines the extent to which the resource name is visible and available to other processes; the same name string in two different domains would indicate two different resources. By default OpenVMS includes the UIC group of the process creating the resource as part of a resource name. Processes belonging to other UIC groups can only share such a resource by joining the resource domain.

Table A.1 OpenVMS Facility Codes

| *Facility* | *Code* | *Section* |
|---|---|---|
| OpenVMS executive | SYS$ | Section A.2 |
| $MOUNT system service | MOU$ | Section A.3 |
| $DISMOU system service | DMT$ | Section A.4 |
| File system (Files-11 XQP) | F11B$ | Section A.5 |
| Record Management Services (RMS) | RMS$ | Section A.6 |
| Image activator and Install utility | INSTALL$ | Section A.7 |
| NETACP–VMScluster alias | CLU$ | Section A.8.1 |
| NETACP–proxy | NET$ | Section A.8.2 |
| Job controller | JBC$ | Section A.9 |
| Queue manager | QMAN$ | Section A.9 |
| System Management (SYSMAN) utility | SMISERVER$ | Section A.10 |

To share resources throughout a VMScluster system independent of UIC, many OpenVMS facilities specify the Enqueue Lock Request ($ENQ) system service flag LCK$V_SYSTEM. A process not in kernel or executive mode requires the SYSLCK privilege to specify this flag. The flag causes the resource to be created in the system domain; OpenVMS omits the UIC group from the resource name. Such a resource is sometimes characterized as being systemwide. This appendix refers to the name space of these resources and locks as the system domain.

Other OpenVMS facilities, such as the job controller, require a process on each VMScluster node. The processes are created in a controlled fashion and belong to the same UIC group. Each process synchronizes access to private structures and files using a protocol shared by its counterparts on other nodes. These processes do not use the LCK$V_SYSTEM flag in their lock requests; thus, their resources and locks are available only to members of the same resource domain. This appendix refers to the name space of such resources and locks as a UIC-based domain.

The lock manager deallocates a resource block (RSB) when its last lock is dequeued. Locks are dequeued and lock blocks deallocated when their creating process is deleted. To guarantee the survival of an important lock so that its resource block and especially its value block remain available, an OpenVMS facility queuing a lock can dissociate the lock from process context by declaring it to be system-owned rather than process-owned, using the LCK$V_CVTSYS flag in the lock request. A system-owned lock's name space can be a UIC-based resource domain or the system resource domain; the name space is distinct from the ownership.

A system-owned lock has no associated process context. Because asynchronous system trap (AST) delivery requires process context, system-owned locks cannot include completion or blocking ASTs. Since ASTs are used in deadlock detection, system-owned locks are not allowed to exist in any

state other than granted. The use of system-owned locks is reserved to Digital. Any other use is strongly discouraged by Digital and completely unsupported.

A parent resource is used to create a logical lock grouping (a resource tree) or, in the case of the System ID lock, to restrict resource mastership to a particular node (see Section A.2.1).

A significant aspect of OpenVMS lock use is the resource's value block; many OpenVMS locking protocols use value blocks to pass limited amounts of critical information among participants. A lock request that specifies a value block receives a private copy of the value block when the lock is granted or when it is converted to a mode at least as restrictive as its current grant mode. Thus, a locking protocol might specify the conversion of a NL mode lock to NL mode only to obtain an updated value block. The owner of a PW or EX mode lock can write the value block when it dequeues the lock or converts the lock to a mode as restrictive as, or less restrictive than, the current mode.

Other significant aspects of OpenVMS lock use include the presence or absence of a blocking AST and the name of any symbol used to locate the lock or define the resource name. Blocking ASTs are described in Chapter 4.

Every lock description in this appendix begins with a table of the lock's significant attributes.

## A.2 EXECUTIVE LOCK USE

### A.2.1 System ID Lock

| | |
|---|---|
| Resource name string | "SYS$SYS_ID" + SCSSYSTEMID |
| Symbol | EXE$GL_SYSID_LOCK |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | None; system-owned |

The System ID lock guarantees a unique identity for each VMScluster node by enforcing the requirement that the SYSGEN parameter SCSSYSTEMID be unique within the VMScluster system.

During system initialization every OpenVMS system requests an EX lock on a resource whose name is based on its own SCSSYSTEMID. Since SCSSYSTEMID is required to be unique in a VMScluster system, the lock should be granted immediately. If the lock request is successful, the numeric lock ID is stored in the cell EXE$GL_SYSID_LOCK. If the lock request fails,

an identical SCSSYSTEMID exists in the VMScluster system. An error message is generated and further system initialization is prevented.

Since each VMScluster node builds and locks a unique resource, the System ID lock is always mastered on the local system. Therefore, any sublock of the System ID lock is mastered on the local system. Many OpenVMS facilities take advantage of this feature and use the lock ID in EXE$GL_SYSID_LOCK as a parent for locks to be mastered locally and for locks whose range is limited to a specific VMScluster node rather than to the entire VMScluster system.

### A.2.2 Set Time Lock

| | |
|---|---|
| Resource name string | "SYS$CWSETIME" |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

The Set Time lock serializes concurrent SET TIME/CLUSTER operations. The image that runs in response to this Digital command language (DCL) command acquires an EX mode lock on the resource SYS$CWSETIME. Even if more than one process enters the SET TIME/CLUSTER command simultaneously, only one process acquires the lock while the others wait. Therefore, the same time value is broadcast to all VMScluster nodes during this interval. When the owning process releases the lock, a waiting process may acquire it and broadcast its own time value. This mechanism ensures that time is broadcast consistently across all VMScluster nodes.

### A.2.3 Device Lock

| | |
|---|---|
| Resource name string | "SYS$" + allocation class device name |
| Symbol | UCB$L_LOCKID |
| Modes of acquisition | CR, PW, EX |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | Yes |
| Blocking AST | None |
| Process context | None; system-owned |

Device locks propagate the standard OpenVMS properties for device allocation throughout a VMScluster system. They manage the availability of devices visible clusterwide. The Deallocate Device ($DALLOC), Assign I/O Channel ($ASSIGN), Mount Volume ($MOUNT), Dismount Volume ($DISMOU), and Deassign I/O Channel ($DASSGN) system services, among others, acquire and release Device locks either directly or using the routines IOC$LOCK_DEV and IOC$UNLOCK_DEV in module IOSUBPAGD.

A VMScluster node actually has at most one Device lock queued per device, with a resource name based on the allocation class device name as returned by the Get Device/Volume Information ($GETDVI) system service argument DVI$_ALLDEVNAM. Its lock ID is stored in the device's unit control block (UCB) at UCB$L_LOCKID.

A Device lock is queued or converted for a device visible to the VMScluster system when the device is explicitly allocated, when the $MOUNT system service implicitly allocates the device for a private mount request, when the $MOUNT system service must ensure that the device is available and not allocated for a shareable request, when the $ASSIGN system service creates the first channel to a device that is available clusterwide, and through other code paths as well.

The lock mode varies depending on the operation and its arguments:

- At device allocation, an EX mode lock is requested.
- For a private mount, the $MOUNT system service requests an EX mode lock.
- For a system or group mount, the $MOUNT system service initially requests a PW mode lock with the LCK$V_NOQUEUE flag. If the device is already allocated or mounted privately, an EX mode lock exists, the PW request fails, and the $MOUNT system service returns an error. If the device is already mounted in a shareable fashion by any other VMScluster nodes, only CR mode locks exist. The PW mode lock is granted and eventually converted to CR mode.
- The $ASSIGN system service requests a CR mode lock.

The value block of a Device lock contains such information as a device's mount state, protection, ownership, shadow set membership, and write lock state. It coordinates these attributes across the cluster.

System services like $DALLOC, $DASSGN, and $DISMOU call the routine IOC$UNLOCK_DEV to dispose of the Device lock correctly:

- If the device remains allocated by a process, the lock is not dequeued until device deallocation.
- If channels remain open to a dismounted device, the lock is converted to CR mode and eventually dequeued during the closing of the last channel.
- Otherwise, the lock is dequeued.

## A.3 $MOUNT LOCK USE

The $MOUNT system service establishes a lock to guard against concurrent mount requests for a particular device or volume from the local node or from other VMScluster nodes. In addition, it acquires the system-owned Device lock (see Section A.2.3) to synchronize clusterwide device access with the $DISMOU, $DALLOC, and $ASSIGN system services, among others. The $MOUNT system service uses the file system's Volume Allocation lock (see Section A.5.1) to synchronize its accesses to mounted volumes with those of the file system. It compares the mount context information in the value blocks of the Device lock and the Volume Allocation lock to ensure that volume labels are unique within a VMScluster system.

### A.3.1 Label Lock

| | |
|---|---|
| Resource name string | "MOU$" + CSID or zero + volume label as specified in $MOUNT argument |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

The Label lock serializes shareable mount requests for the same volume from multiple processes on the same system. In response to a request to mount a volume to be shared, for example, among all UIC-group members, the $MOUNT system service requests this lock in EX mode. It includes the VMScluster system ID (CSID) to make the Label lock node-specific. If the system is not a VMScluster node, a CSID of zero is used.

For a shareable mount request, the $MOUNT system service searches the local I/O database to ensure that no other volume has been mounted with the same volume label and shareability. It holds the Label lock for the duration of local mount processing to prevent other processes running on the same node from trying to mount the same volume on other devices.

The $MOUNT system service cannot use either of the other two locks involved in mount processing to accomplish that purpose. The Mount Device lock (see Section A.3.2) is based on device name, not volume label, so its use would not detect simultaneous attempts to mount a volume with the same label on different devices. Neither would the use of the system-owned Device lock, which is only acquired for devices available clusterwide.

The $MOUNT system service does not acquire the Label lock for a private mount request, because process-private use of a particular volume name cannot conflict with that of any other use.

### A.3.2 Mount Device Lock

| | |
|---|---|
| Resource name string | "MOU$" + allocation class device name |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

The Mount Device lock synchronizes simultaneous mount requests for the same device. The $MOUNT system service first locates the device to be mounted and reserves it with the system-owned Device lock. It then attempts to acquire the Mount Device lock in EX mode. If the Mount Device lock cannot be immediately acquired because another $MOUNT request is proceeding concurrently on the same device, the $MOUNT system service releases the system-owned Device lock and queues for the Mount Device lock in EX mode. When the Mount Device lock is granted, the $MOUNT system service releases it and repeats its attempt to acquire the system-owned Device lock.

Thus, mount attempts in a VMScluster wait for the Mount Device lock rather than the system-owned Device lock when the system-owned Device lock is not immediately available. A process cannot wait for a system-owned lock.

The Mount Device lock's resource name is based on the allocation class device name as returned in the $GETDVI system service argument DVI$_ALLDEVNAM.

## A.4 $DISMOU LOCK USE

### A.4.1 Dismount Lock

| | |
|---|---|
| Resource name string | "DMT$" + allocation class device name |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |

| | |
|---|---|
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

The $DISMOU system service acquires an EX mode Dismount lock to synchronize simultaneous dismount requests for the same volume from processes on the local system and on other VMScluster nodes. The Dismount lock's resource name is based on the allocation class device name returned in the $GETDVI system service argument DVI$_ALLDEVNAM.

In addition to the Dismount lock, the $DISMOU system service acquires, converts, and releases the system Device lock to update the value block. The $DISMOU system service also dequeues file system locks for Files-11 volumes.

## A.5 FILE SYSTEM LOCK USE

The file system uses locks to arbitrate access to volumes and files as well as access to local cache structures and their contents. In a VMScluster system, all locks described in this section are necessary for proper synchronization. In a stand-alone system, the volume locks and File Serialization lock are required for synchronization of local processes, but the File Access Arbitration and Cache locks are unnecessary and unused.

### A.5.1 Volume Allocation Lock

| | |
|---|---|
| Resource name string | "F11B$v" + VCB$T_VOLCKNAM or RVT$T_VLSLCKNAM |
| Symbol | VCB$L_VOLLKID or RVT$L_STRUCLKID |
| Modes of acquisition | CR, PW |
| Domain | System domain |
| Access mode | Kernel |
| Parent | EXE$GL_SYSID_LOCK if private mount |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | None; system-owned |

The Volume Allocation lock synchronizes volume space allocation by coordinating access to the storage and file header bitmaps. Each volume has a unique volume allocation resource name. Every VMScluster node acquires a PW mode lock on that resource when it mounts the volume through the $MOUNT system service. The resource name string is based on the contents of VCB$T_VOLCKNAM or, for volume sets, the volume set name contained in relative volume table (RVT) field RVT$T_VLSLCKNAM:

- For a privately mounted volume, the resource name string is based on the name of the system issuing the $MOUNT request and that system's UCB address for the device. The Volume Allocation lock is a sublock of the lock ID stored in EXE$GL_SYSID_LOCK.
- For a shareable native volume, the resource name string is based on the volume label.
- For a shareable native volume set, the resource name string is the volume set name.

The naming convention for shareable volumes guarantees that volume labels are unique in a VMScluster system.

The Volume Allocation lock is converted to CR mode by each VMScluster node when the $MOUNT system service completes. The lock ID is stored in VCB$L_VOLLKID (or, for volume sets, RVT$L_STRUCLKID). This lock is held in CR mode for as long as the volume remains mounted. The $DISMOU system service dequeues it. In addition, any code path that allocates or deallocates space on the volume (that is, accesses the index file bitmap or the storage bitmap) acquires an additional lock in PW mode. This is compatible with the CR mode locks but would block another PW mode lock; thus, it allows multiple readers but only one writer.

### A.5.2 Volume Blocking Lock

| | |
|---|---|
| Resource name string | "F11B$b" + VCB$T_VOLCKNAM or RVT$T_VLSLCKNAM |
| Symbol | VCB$L_BLOCKID or RVT$L_BLOCKID |
| Modes of acquisition | CR, PW, EX |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | None |
| Blocking AST | Yes |
| Process context | System-owned and process-owned |

The Volume Blocking lock enables exclusive access to a volume by utilities such as the Analyze/Disk Structure utility. Its lock ID is stored in VCB$L_BLOCKID or RVT$L_BLOCKID as appropriate.

The Volume Blocking lock is normally held by all nodes in CR mode. To lock the volume, a utility requests an EX mode process-owned lock on the resource. This causes a blocking AST to be delivered to each VMScluster node holding a Volume Blocking lock, including the node on which the utility is executing. The lock manager dispatches to the blocking AST procedure at IPL$_SCS while holding the SCS spinlock.

The blocking AST procedure clears VCB$L_BLOCKID. The field VCB$L_ACTIVITY reflects the state of the Volume Blocking lock. The field is initialized to 1, and the volume remains usable as long as the field is odd. Normal file system activity on the volume increments the VCB$L_ACTIVITY count by 2 and decrements it by 2 on completion. The blocking AST procedure decrements VCB$L_ACTIVITY by 1, making its value even, and thus blocks further file system requests for the volume. If this decrement of VCB$L_ACTIVITY brings its value to zero, the routine requests a kernel mode AST to dequeue the Volume Blocking lock from the context of the swapper process. Otherwise, as each outstanding file system request completes, it decrements VCB$L_ACTIVITY by 2. When VCB$L_ACTIVITY eventually falls to zero, the completing file system request dequeues the CR mode lock. This allows the EX mode lock to be granted so that the operation requiring exclusive access can proceed.

After the EX mode lock is released, the next file system request reacquires the Volume Blocking lock before accessing the volume.

### A.5.3 File Access Arbitration Lock

| | |
|---|---|
| Resource name string | "F11B$a" + volume lock name + FCB$L_LOCKBASIS |
| Symbol | FCB$L_ACCLKID |
| Modes of acquisition | All |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | None; system-owned |

The file system provides access arbitration for files; that is, users can open files for read or write operations and specify whether other users may open the file concurrently. The Access Arbitration lock extends the scope of file arbitration to be clusterwide. Its resource name string uniquely identifies a particular file by including the volume lock name from VCB$T_VOLCKNAM or RVT$T_VLSLCKNAM, and the file's ID number and relative volume number from the file control block (FCB) field FCB$L_LOCKBASIS. Each VMScluster node on which at least one process has that file open holds one system-owned Access Arbitration lock. Each lock represents the state of all accesses to the file from a given node. Thus, a VMScluster node acquires the lock in the most restrictive mode in which any of its local processes have opened the file.

The Access Arbitration lock's blocking AST synchronizes access to its associated FCB, which contains information from the file header, such as protection and size. Each VMScluster node accessing the file has an FCB and an Access Arbitration lock for the file. When a node alters an FCB in its memory, it also requests an EX mode Access Arbitration lock. This causes execution of the blocking AST procedure on every node accessing the file, causing each to mark its FCB as stale. Each node rebuilds its FCB on the next local access.

### A.5.4 File Serialization Lock

| | |
|---|---|
| Resource name string | "F11B$s" + FCB$L_LOCKBASIS |
| Symbol | None |
| Modes of acquisition | NL, PW |
| Domain | System domain |
| Access mode | Kernel |
| Parent | VCB$L_VOLLKID or RVT$L_STRUCLKID |
| Value block | Yes |
| Blocking AST | None |
| Process context | System-owned when NL; process-owned when PW |

A File Serialization lock synchronizes access to a file on a particular volume. The file's ID number and relative volume number from FCB$L_LOCKBASIS make up the resource name string; its parent is a Volume Allocation lock. The file system, running in local process context, requests a File Serialization lock in PW mode for the duration of a single file operation, such as create, extend, or truncate. A process must hold the lock before accessing a file header or associated data in a file system cache. The lock value block contains two sequence numbers, one for the file header and one for associated data.

Upon completion of the file operation, the file system converts the lock to a NL mode system-owned lock, rewrites the sequence numbers into the value block, and records them in a cache descriptor. The system-owned lock is maintained until the cache entry is removed from cache or reused as described in the following paragraph.

If a process on this VMScluster node requests a subsequent access to the file, the file system acquires the File Serialization lock in PW mode and obtains the sequence numbers in its value block. It compares the sequence numbers to the stored values in the cache descriptor. If the values match, the cached information is still accurate. Otherwise, another VMScluster node acquired a PW mode lock while this node held a NL mode lock, and performed

a file operation that updated a sequence number. The information in the local cache is no longer accurate and must be reread.

### A.5.5 Cache Locks

Cache locks synchronize access to the per-volume caches that exist on each VMScluster node for each mounted volume: the file ID cache, extent cache, and disk quota cache. This section describes the general mechanism used to cause each VMScluster node in turn to flush a particular cache's contents to disk. Sections A.5.5.1, A.5.5.2, and A.5.5.3 describe the individual locks. The file system flushes all per-volume caches when a volume is dismounted. It flushes an individual cache when the cache becomes full, when a privileged user attempts to access the associated cache disk file directly, when one VMScluster node's cache is empty, and on similar occasions.

Each cache type has a defined cache flush resource name. Each VMScluster node that mounts a volume acquires a lock on each of the three cache flush resources for the volume. These locks are normally system-owned and held in PR mode.

To flush cache entries back to disk, a VMScluster node writes its own cache back under the protection of a PW mode Volume Allocation lock. It then marks the particular cache invalid, lowers the system-owned PR mode Cache lock to NL mode, and lowers the Volume Allocation lock back to CR mode, rewriting the value block. The node then requests an additional process-owned CW mode lock on the cache flush resource.

This causes blocking AST delivery to all other VMScluster nodes holding PR mode Cache locks. Since these are system-owned locks, the lock manager dispatches to the blocking AST routine at IPL$_SCS while holding the SCS spinlock. The AST parameter identifies which volume and cache to flush. Each blocking AST routine uses an AST control block (ACB) built into the cache data structure to deliver an AST to the CACHE_SERVER process. The CACHE_SERVER process requests the Queue I/O ($QIO) system service with the function code IO$_ACPCONTROL and a parameter identifying the device and cache.

The file system, running in the context of the CACHE_SERVER process, requests a PW mode Volume Allocation lock on the appropriate volume. Only one VMScluster node's request for this lock is granted; the other nodes wait. The node that successfully acquires the Volume Allocation lock flushes its cache, marks the cache invalid, and lowers its PR mode Cache lock to NL mode. Next it converts the Volume Allocation lock back to CR mode, rewriting the value block. One waiting Volume Allocation lock request from another node is granted, and that node flushes its cache. This sequence is repeated until each node in turn has flushed its cache.

While the cache flush is in progress, the cache is marked invalid. If the file system accesses it and finds it invalid, the file system requests conversion of the NL mode Cache lock back to PR mode.

When the last VMScluster node completes and converts its PR mode Cache lock to NL mode, the original CW mode request is granted and immediately dequeued, and the cache flush is complete.

### A.5.5.1 File ID Cache Lock

| | |
|---|---|
| Resource name string | "F11B$c" + lock basis of INDEXF.SYS |
| Symbol | VCA$L_FIDCLKID |
| Modes of acquisition | NL, CW, PR |
| Domain | System domain |
| Access mode | Kernel |
| Parent | VCB$L_VOLLKID or RVT$L_STRUCLKID |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | None; system-owned |

Each VMScluster node maintains its own cache of available file headers for each mounted volume. This cache is filled primarily by file deletion on the local node. Any file identification numbers (FIDs) held in the cache are still marked "in-use" in the disk file number bitmap. A cache flush requires each VMScluster node to write all entries in its local cache back to the file number bitmap on disk. The File ID Cache lock arbitrates this cache flush across the VMScluster system, as described in Section A.5.5.

### A.5.5.2 Extent Cache Lock

| | |
|---|---|
| Resource name string | "F11B$c" + lock basis of BITMAP.SYS |
| Symbol | VCA$L_EXTCLKID |
| Modes of acquisition | NL, CW, PR |
| Domain | System domain |
| Access mode | Kernel |
| Parent | VCB$L_VOLLKID or RVT$L_STRUCLKID |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | None; system-owned |

Each VMScluster node maintains its own cache of available disk space for each mounted volume. Any disk blocks held in this cache are still marked "in-use" in the disk storage allocation bitmap. A cache flush requires each VMScluster node to write all entries in its local cache back to the storage

allocation bitmap on disk. The Extent Cache lock arbitrates this cache flush across the VMScluster system, as described in Section A.5.5.

### A.5.5.3 Disk Quota Cache Lock

| | |
|---|---|
| Resource name string | "F11B$c" + lock basis of QUOTA.SYS |
| Symbol | VCA$L_QUOCLKID |
| Modes of acquisition | NL, CW, PR |
| Domain | System domain |
| Access mode | Kernel |
| Parent | VCB$L_VOLLKID or RVT$L_STRUCLKID |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | None; system-owned |

If disk quotas are enabled for a volume, a disk quota cache and Disk Quota Cache lock are created when the volume is mounted.

Each VMScluster node maintains its own cache of quota entries. It must sometimes flush all valid entries back to disk, for example, before dismounting the device. The Disk Quota Cache lock arbitrates this cache flush across the VMScluster system, as described in Section A.5.5.

### A.5.5.4 Quota Cache Entry Lock

| | |
|---|---|
| Resource name string | "F11B$q" + VCB$T_VOLCKNAM or RVT$T_VLSLCKNAM + quota record UIC |
| Symbol | VCA$L_QUOLKID |
| Modes of acquisition | CR, PW, EX |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | None; system-owned |

To acquire a user's quota information, a VMScluster node queues a PW mode system-owned Quota Cache Entry lock. On the first access to a specific quota cache entry, the user's quota information is read from disk into a cache block. The dynamic portion of the user's quota information is shared among VMScluster nodes through the value blocks of the Quota Cache Entry locks for that user.

When another VMScluster node needs the same user's quota information, it requests its own PW mode system-owned Quota Cache Entry lock. This request causes a blocking AST to be delivered to the original lock owner. The

blocking AST procedure, running in the swapper's process context, marks the local cache entry invalid. It converts the PW mode lock to CR mode, updating the value block with the shared quota information. The other node's PW mode lock request is granted, and it receives this quota information from the value block.

An EX mode lock on a quota cache entry causes VMScluster nodes to remove the entry from the quota cache. This is used when a quota record is deleted.

## A.6 RMS LOCK USE

Record Management Services (RMS) uses lock management system services to protect files and records. When a file is accessed in a shareable fashion with write access allowed, RMS uses locks to coordinate the actions of the file sharers. The locks that it requests depend on a file's organization, the presence of global buffers, and numerous file-sharing and record-locking options specified by the user application. This section describes some of the more common RMS locks, sometimes in a simplified manner. It does not include locks used for RMS journaling.

RMS runs in process context and maintains private data structures in process space. It requires a file access block (FAB) for each initial access (open) of a file and a record access block (RAB) for each stream connected to a FAB. It creates internal copies of FABs and RABs called IFABs and IRABs (in data structures named IFB$ and IRB$) as well as many internal structures mentioned briefly in this appendix.

A process can optionally open a file multiple times (with multiple FABs). The term *accessor*, as used in this appendix, indicates an entity in process context that has opened the file; for example, a file opened twice by process A and once by process B would have three accessors. Additionally, each accessor can optionally connect multiple record streams to the file (multiple RABs to each FAB). RMS therefore must synchronize file access among accessors and record streams from the same process through a variety of mechanisms. It must also synchronize independent processes sharing a file on a local system or in a VMScluster.

RMS transfers a bucket of data on a process's behalf from a file into a buffer in memory. An RMS local buffer is mapped in process space and is available to only one process. A global buffer is mapped in system space within an OpenVMS global section and can be shared by any process on the system. Global buffers, however, cannot be shared by processes on different VMScluster nodes.

RMS performs some functions that affect the internal file structure, such as altering the end-of-file marker; some functions that affect internal bucket or buffer structure or contents; and some functions that affect only record contents. It uses locks of different scope to protect these different functions.

RMS enforces a strict hierarchy in the acquisition of locks to ensure that deadlocks do not occur. A user application has no direct control over RMS locks other than record locking. Thus, for locks other than Record locks, RMS can safely specify the LCK$V_NODLCKWT and LCK$V_NODLCKBLK flags in its $ENQ system service requests. Chapter 4 gives more information on these flags.

With the exception of Record locks, RMS holds locks in restrictive lock modes only for the duration of an RMS service request. To operate more efficiently and to preserve lock value block information, especially the sequence number, RMS typically converts a lock to NL mode rather than releasing it altogether.

### A.6.1 File Lock

| | |
|---|---|
| Resource name string | "RMS$" + file ID + device name |
| Symbol | SFSB$L_LOCK_ID, IFB$L_PAR_LOCK_ID |
| Modes of acquisition | NL, CR, PR, PW, EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |
| Value block | Yes |
| Blocking AST | Sometimes |
| Process context | Process-owned |

The RMS File lock synchronizes clusterwide, interprocess, and intraprocess access to a file's RMS metadata rather than to the actual blocks of data within the file. The metadata includes such items as the end-of-file marker, the length of the longest record, and the number of the highest allocated block. All these items are cached in the File lock's value block. Much of this data is static; the most volatile portion is the end-of-file marker, particularly for sequential files.

The File lock resource name string uniquely identifies a file and the device where it resides. RMS builds the File lock resource name string from the six-byte file identifier plus the device identifier returned in the $GETDVI system service argument DVI$_DEVLOCKNAM. The device identifier is normally the mount type code followed by the volume name from VCB$T_VOLCKNAM or RVT$T_VLSLCKNAM. The mount type code is 1 for a privately mounted device or 2 for a device mounted in a shareable fashion.

When a process opens a file, it tells RMS how it wishes to access the file and the type of access that it will allow to other accessors. RMS creates a File lock for a file that is shared and to which write access is allowed. It keeps this File lock until the process closes the file. To enhance performance, a process acquires only one File lock when it opens a particular file, no matter how

many RAB streams it connects to the file. The streams share the File lock in a manner that requires as few lock conversions as possible.

RMS uses the following lock modes for the File lock:

- A NL mode lock indicates interest in the file.
- To increase a value block item, for instance, to extend a file by incrementing the end-of-file marker, it converts the File lock to a PW mode lock with a blocking AST.
- To decrease a value block item, for instance, to truncate a file by decrementing the end-of-file marker, it converts the File lock to an EX mode lock with a blocking AST.
- To read the most current values from the value lock, it converts the File lock to PR mode.
- When only a lower bound is needed for an item in the value block, it converts the File lock to CR mode.

RMS initially requests a NL mode File lock. It specifies LCK$V_VALBLK to include a value block and LCK$V_EXPEDITE to ensure that the lock is granted despite any queued conversion requests from other processes also accessing the file.

If the returned value block is uninitialized, no other accessors exist for the file, and RMS converts the File lock to EX mode and initializes the value block itself from the file header. Otherwise, the returned value block contains data. If an EX or PW mode File lock exists, however, the returned value block data is not current; the holder of the EX or PW mode lock is updating the value block and will rewrite it when releasing its lock.

Therefore, to obtain the current value block, RMS converts its NL mode lock to a PR mode lock. This causes blocking AST delivery, and the holder of the EX or PW mode lock converts its lock to NL mode, updating the value block. The PR mode lock can then be granted, and it receives a copy of the newly updated value block.

In some cases, only a lower bound is needed for a value block field. As a performance enhancement, RMS converts its File lock to CR mode. If another accessor is updating the value block to increased values, it holds a PW mode File lock. CR mode is compatible with PW mode, so it is granted. This avoids blocking AST delivery and minimizes lock overhead when the value block contains valid lower bound data.

CR mode is not, however, compatible with EX mode, used for decreasing values. Therefore, when the value block does not contain valid lower bound data, an EX mode lock exists. Its blocking AST is delivered, and the lock is converted to NL mode, updating the value block. The CR mode lock can then be granted, and it receives a copy of the newly updated value block.

RMS stores the lock ID of the File lock in IFB$L_PAR_LOCK_ID. It uses the File lock as the parent lock of a file's Bucket locks (see Section A.6.3) and of its Record locks (see Section A.6.5).

RMS creates a shared file synchronization block (SFSB) in process space for each accessor using a File lock. The SFSB describes the accessor's File lock: its resource name, lock ID, and other items. The SFSB contains the local copy of the lock value block contents. The various file organizations treat its fields differently:

- SFSB$L_EBK contains the virtual block number of the last block of the file.
- SFSB$W_FFB contains the next available byte in the last block of the file.

  SFSB$L_EBK and SFSB$W_FFB together constitute the end-of-file (EOF) marker. They are of particular importance for sequential files; for relative and indexed files, SFSB$W_FFB is set to zero and SFSB$L_EBK is static.
- SFSB$L_HBK contains the highest allocated block of the file.
- SFSB$W_LRL contains the longest record written to the file thus far, for sequential files. This field is zero for relative and indexed files, unless the record format is fixed length.
- SFSB$B_MBC contains the bucket size for a sequential file.
- SFSB$B_LKVB_VER contains the RMS locking protocol version.

An SFSB also contains status bits used to support File lock sharing by multiple record streams associated with one accessor (when multistreaming is selected).

| *Bit Field Name* | *Meaning if Set* |
|---|---|
| SFSB$V_WANTED | File lock is wanted by another stream |
| SFSB$V_IN_PROGRESS | Request for File lock is being processed |
| SFSB$V_KEEP_LOCK | File lock is owned |

### A.6.2 EOF Lock

| | |
|---|---|
| Resource name string | "APPENDER" |
| Symbol | SFSB$L_EOF_LOCK_ID |
| Modes of acquisition | NL, PW |
| Domain | System domain |
| Access mode | Executive |
| Parent | IFB$L_PAR_LOCK_ID |
| Value block | No |
| Blocking AST | Sometimes |
| Process context | Process-owned |

To avoid thrashing caused by lock contention, a stream that stalls does not release the File lock or any other locks that it holds. For instance, when RMS performs an I/O operation on a process's behalf to read a virtual block from a file on disk into a bucket, it does not release the bucket lock while waiting for information to be returned from the disk. Thus, to simplify the implementation and prevent deadlocks, a stream that holds the File lock in an elevated access mode is not allowed to acquire any additional locks.

However, updating the value block fields for sequential files is a complex operation and requires additional synchronization. To provide this synchronization, the RMS EOF lock is used in conjunction with the File lock. The EOF lock is normally held in NL mode. At the start of a complex operation that modifies a value block field, the EOF lock must be converted to PW mode. The File lock is next acquired in PR mode to read the value block, then converted to NL mode.

Holding the EOF lock, the stream performs its file operations, possibly involving multiple bucket locks and I/Os to the disk. The File lock value block fields can be depended upon over the course of these I/O operations, since any stream desiring to change them would first contend for the EOF lock. When the file operation is complete, the stream requests the File lock in PW mode to update the value block. It releases first the File lock, then the EOF lock.

RMS maintains EOF lock information in the same SFSB as its parent File lock. The SFSB contains the EOF lock ID and status bits used to support EOF lock sharing by multiple record streams associated with one accessor (when multistreaming is selected).

| *Bit Field Name* | *Meaning if Set* |
|---|---|
| SFSB$V_EOF_WANTED | EOF lock is wanted by another stream |
| SFSB$V_INUSE | EOF lock is in use by this accessor |
| SFSB$V_TAKEN | EOF lock is held by this accessor |

### A.6.3 Bucket Lock

| | |
|---|---|
| Resource name string | Bucket virtual block number |
| Symbol | BLB$L_LOCK_ID |
| Modes of acquisition | NL, PW, EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | IFB$L_PAR_LOCK_ID |
| Value block | Yes |
| Blocking AST | Sometimes |
| Process context | System-owned and process-owned |

RMS Bucket locks ensure the integrity of buckets held in local or global buffers. The resource name string of the Bucket lock identifies the first virtual block number of the bucket data within the file. Because RMS must acquire an EX mode process-owned Bucket lock for an accessor before it can read or write the bucket, it can maintain a consistent picture of the bucket contents clusterwide.

RMS reads a bucket into either an I/O buffer in a process's address space or an RMS global I/O buffer in an OpenVMS global section in system space.

It protects buckets in both locations through Bucket locks. One difference between Bucket locks for buckets in local and in global buffers is the parent lock. The IFB$L_PAR_LOCK_ID cell identifies the Bucket lock's parent: the File lock for a local buffer, or the Global Buffer Master lock (see Section A.6.4.1) if global buffers are being used.

RMS I/O buffers are a limited commodity; both local and global buffers are used and reused under the control of an RMS cache replacement algorithm. RMS maintains information about local buffer entries that are in use through buffer descriptor blocks (BDBs) and buffer lock blocks (BLBs). Before a process fills a local buffer from a bucket, it obtains a BDB and, if RMS locking is being performed, a BLB. It then acquires an EX mode, process-owned Bucket lock.

RMS stores information regarding the Bucket lock in the BLB, including the lock ID, an identifier for the record stream that owns the lock, the lock status block, the lock value block, the lock resource name, and the associated BDB address. The BDB contains, among other items, the actual address of the buffer and a saved clusterwide sequence number for the bucket that currently resides in the buffer.

The Bucket lock value block contains a sequence number for the bucket. A process must own the Bucket lock in EX mode before modifying the bucket so that it can increment the sequence number in the lock. This invalidates any buffer containing an earlier version of the bucket. For example, an accessor might have a version (possibly an outdated version) of a bucket in a local buffer, with an associated NL mode Bucket lock, BLB, and BDB. To reaccess the bucket, RMS converts the NL mode Bucket lock to EX mode, rereading the value block. RMS compares the new sequence number from the lock value block with the buffer's saved sequence number, stored in the BDB. If the sequence numbers do not match, this buffer contains an outdated copy of the bucket and RMS rereads the bucket from disk. If the accessor subsequently modifies the bucket, it increments the sequence number. When it completes its bucket access, it converts the lock from EX to NL mode, rewriting the value block with the updated sequence number.

RMS maintains a NL mode Bucket lock on a bucket as long as that bucket is in a local or global buffer cache. This preserves the bucket's lock value block and thus its sequence number. One NL mode lock is required per copy of the bucket. Thus, each process that has a copy of a bucket in a local RMS I/O buffer maintains its own NL mode Bucket lock until it reuses that local buffer for a different bucket. For a bucket in a global buffer, however, one copy of the bucket in memory is shared by any interested process on the system. In this case, only one NL mode lock is required per VMScluster node to preserve the bucket's sequence number.

The first accessor of a bucket in a global buffer converts its Bucket lock to a NL mode system-owned Global Buffer Backing lock (see Section A.6.4.3)

when it completes its operation on the bucket. Subsequent accessors merely dequeue their Bucket locks.

An exception to this local conversion to NL mode is the case of a deferred write of modified buckets. For a deferred write, RMS converts the lock to a PW mode lock with an associated blocking AST. When another accessor of the file wants to use the modified bucket, its lock request triggers the execution of the blocking AST procedure, which writes the modified bucket. If no other accessor requests the modified bucket, RMS eventually writes the bucket and dequeues the Bucket lock when cache replacement dictates that the buffer should be reused for another bucket.

### A.6.4 Locks Associated with Global Buffers

To minimize I/O operations, RMS can share buffers among multiple accessors of the same file. It maintains these global buffers in system space, within an OpenVMS global section. A file using global buffers has one such global section on each VMScluster node from which a process accesses the file. When the first process on a VMScluster node opens a file that uses global buffers, RMS creates the file's global section in that node's memory.

RMS constructs the name of the file's global section by appending the hexadecimal address of the file's FCB to the string RMS$. Any accessor subsequently opening the file in the same memory space shares the same FCB and thus constructs the same global section name and maps to the existing global section.

Each global buffer global section contains a global buffer header (GBH), a global buffer descriptor (GBD) for each global I/O buffer within the section, and the global buffers themselves. The actual data resides in buckets within the global buffers.

The GBH describes the global section and its locks. It contains the size of the global section, the access count, and the Global Buffer Master lock ID at offset GBH$L_LOCK_ID, among other information.

One GBD exists for each global buffer in the global section. A global buffer's GBD contains the lock ID of the buffer's Global Buffer Backing lock in the field GBD$L_LOCK_ID, the lock sequence number, the offset to the buffer within the global section, and similar information.

RMS maintains a section's GBDs in an interlocked queue ordered by the VBN of the bucket currently residing in the GBD's associated buffer. The head of the GBD queue is in the GBH, at offset GBH$L_GBD_FLINK.

The Global Buffer Section (GBS) lock serializes access to the global buffer header and thus to the GBD queue and the global buffer pool (see Section A.6.4.2).

RMS deletes a file's global section when the last accessor of the file on a VMScluster node closes the file.

### A.6.4.1 Global Buffer Master Lock

| | |
|---|---|
| Resource name string | "RMS$" + file ID + device name |
| Symbol | GBH$L_LOCK_ID, IFB$L_PAR_LOCK_ID |
| Mode of acquisition | NL |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | None; system-owned |

RMS creates a Global Buffer Master lock only for a file that uses global buffers. The Global Buffer Master lock is a system-owned NL mode version of the File lock (see Section A.6.1).

When an accessor requests shareable write access to a file, RMS creates a File lock. If the file uses global buffers, RMS creates an additional, NL mode File lock and converts it to a system-owned NL mode Global Buffer Master lock when the first record stream is connected. It copies the lock ID of the Global Buffer Master lock to IFB$L_PAR_LOCK_ID, overriding the accessor's File lock as parent of its Bucket locks.

The Global Buffer Master lock's sole purpose is to serve as the parent lock for an accessor's Bucket locks on global buffers. Since a global buffer survives the deletion of processes that use it, the Bucket lock on a global buffer must be backed up with a system-owned lock so that the value block, which maintains the integrity of the bucket, survives. Since a system-owned lock cannot be a sublock of a process-owned lock such as the File lock, a Bucket lock needs a system-owned version of the File lock to act as parent.

RMS dequeues the Global Buffer Master lock when it deletes the global section.

### A.6.4.2 Global Buffer Section Lock

| | |
|---|---|
| Resource name string | "RMS$" + file ID + device name |
| Symbol | GBSB$L_LOCK_ID |
| Modes of acquisition | NL, EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | EXE$GL_SYSID_LOCK |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | Process-owned |

The Global Buffer Section (GBS) lock synchronizes access to a file's global buffer header, the global buffer descriptor queue in the GBH, and thus the global buffer pool. Each VMScluster node accessing the file has a separate global buffer global section for the file. Therefore, a lock guaranteed to be mastered on the local VMScluster node is a more efficient way to serialize global section access, so RMS creates the GBS lock as a sublock of EXE$GL_SYSID_LOCK.

The GBS lock resource name string matches that of the file's corresponding File lock, thus uniquely identifying a device and file in a VMScluster system (see Section A.6.1).

When an accessor connects a record stream to a file that uses global buffers, RMS requests an EX mode GBS lock. The GBS lock remains in EX mode until another accessor sharing the file on the same VMScluster node requests a GBS lock in EX mode, to search the GBD list, for example.

The request triggers blocking AST notification, and the lock holder converts the initial GBS lock to NL mode, allowing the requestor to acquire its own lock.

Before the original lock holder accesses the global section again, it requests the conversion of its NL mode lock back to EX mode.

Therefore, the accessor that most recently examined the global buffer header or searched the global buffer descriptor queue holds the only granted EX mode lock. Every other accessor sharing the globally buffered file on this VMScluster node holds a NL mode lock or is waiting for a new or converted EX mode lock.

When an accessor closes the file, RMS dequeues its GBS lock.

RMS creates a global buffer synchronization block (GBSB) in the P1 space of each accessor holding a NL mode or EX mode GBS lock. The GBSB is similar to the SFSB for the File lock. It maintains information about the lock and the associated global section, including the lock ID at GBSB$L_LOCK_ID. The GBSB also contains the lock value block, the resource name copied from the SFSB, and three status bits:

| *Bit Field Name* | *Meaning If Set* |
|---|---|
| GBSB$V_TAKEN | GBS lock is held in EX mode |
| GBSB$V_INUSE | GBS lock is in use by a record stream |
| GBSB$V_WANTED | GBS lock is wanted by another accessor |

These status bits describe the state of the GBS lock and are treated like the corresponding status bits in the SFSB (see Section A.6.1).

### A.6.4.3 Global Buffer Backing Lock

| | |
|---|---|
| Resource name string | Bucket virtual block number |
| Symbol | GBD$L_LOCK_ID |
| Mode of acquisition | NL |

| | |
|---|---|
| Domain | System domain |
| Access mode | Executive |
| Parent | IFBSL_PAR_LOCK_ID |
| Value block | Yes |
| Blocking AST | None |
| Process context | None; system-owned |

The Global Buffer Backing lock ensures the integrity of buckets contained in global buffers. To guard the integrity of a bucket, its sequence number must be preserved in the lock value block of a Bucket lock. However, a global buffer can contain a bucket that no longer has any current accessors and therefore would have no Bucket locks. Therefore, a system-owned lock must be used to prevent the loss of the bucket's sequence number.

Before reading a bucket from a file into a buffer, an accessor acquires a process-owned Bucket lock. For each global buffer within the global section, the original accessor that stores a bucket in a global buffer converts its process-owned Bucket lock to a NL mode system-owned Global Buffer Backing lock when it completes its access to the bucket. It saves the lock ID in the buffer's GBD at the offset GBD$L_LOCK_ID.

A subsequent accessor of the bucket in this global buffer acquires its own Bucket lock. When it completes its access, it can safely dequeue its Bucket lock, since a Global Buffer Backing lock already exists for the bucket.

RMS also stores the sequence number of a bucket in a global buffer in the buffer's associated GBD. RMS copies the sequence number from the Bucket lock value block of the first accessor of the bucket and updates it for each subsequent accessor. When each accessor obtains its Bucket lock, RMS compares the sequence number in the Bucket lock value block with the saved sequence number in the GBD. If they do not match, RMS rereads the bucket from disk into the global buffer.

RMS dequeues the Global Buffer Backing lock when cache replacement policy dictates that the global buffer should be reused for another bucket or when the global section is deleted.

Since the Global Buffer Backing lock must be system-owned, and system-owned locks cannot be sublocks of process-owned locks, the Global Buffer Master lock was instituted (see Section A.6.4.1).

### A.6.5 Record Lock

| | |
|---|---|
| Resource name string | Record file address |
| Symbol | RLB$L_LOCK_ID |
| Modes of acquisition | CR, PR, PW, EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | SFSB$L_LOCK_ID |

| | |
|---|---|
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

A Record lock coordinates access to a record in a bucket. It is always process-owned and always a sublock of the File lock. RMS builds the Record lock resource name string from the three-word record file address (RFA), which locates the record within the file. The resource name string consists of RFA4, the last of the three words, followed by two bytes of zeros, followed by RFA0, the first word (see the *OpenVMS Record Management Services Reference Manual*).

If a file is opened in a shareable manner with record locking enabled, the following locking options in the user-specified RAB at field RAB$L_ROP determine the RMS lock mode:

| *Bit Field State* | *Lock Mode* |
|---|---|
| RAB$V_REA clear and RAB$V_RLK clear | EX |
| RAB$V_RLK set | PW |
| RAB$V_REA set and RAB$V_RLK clear | PR |
| RAB$V_NLK set | CR |

- EX mode is the default.
- PW mode locks the record for write access, allowing readers at CR mode but no other writers.
- PR mode locks the record for read access, allowing other readers at CR or PR mode but no writers.
- The RAB$V_NLK option temporarily takes a CR mode lock to verify that the record is not locked against reading (in EX mode). These CR mode locks are never returned to the application.

A record stream associates each of its Record locks with a record lock block (RLB). An RLB contains the resource name, an identifier for the owning stream, and the lock status block, including the lock ID. RLBs are linked to the stream's IRAB at the IRB$L_RLB_FLINK/IRB$L_RLB_BLINK queue.

Application record deadlocks are possible because of the control that an application has over its record locking, especially when it selects the manual unlocking (RAB$V_ULK) option.

## A.7 IMAGE ACTIVATOR AND INSTALL UTILITY LOCK USE

### A.7.1 KFE Lock

| | |
|---|---|
| Resource name string | "INSTALL$KNOWN FILE" |
| Symbol | EXE$GQ_KFE_LCKNAM |
| Modes of acquisition | PR, EX |
| Domain | System domain |
| Access mode | Executive |

| | |
|---|---|
| Parent | EXE$GL_SYSID_LOCK |
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

Section A.7.2 describes the use of the KFE lock.

### A.7.2 Install Lock

| | |
|---|---|
| Resource name string | "INSTALL$INSLOCK" |
| Symbol | None |
| Modes of acquisition | PR, PW, EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | EXE$GL_SYSID_LOCK |
| Value block | None |
| Blocking AST | None |
| Process context | Process-owned |

The Install utility manages the known file entry (KFE) list and requires read and write access to it. The image activator system service requires protected read access to the KFE list before opening images. The Install utility, the image activator, and RMS coordinate access to the KFE list through the KFE lock and use the Install lock to provide priority access to image activation.

Each VMScluster node maintains a private KFE list. KFE locks are requested as sublocks of the lock ID stored in EXE$GL_SYSID_LOCK to guarantee that they will be unique to the local node and mastered there. The resource domain is declared to be systemwide because the activation of images and use of the Install utility is not restricted to a single UIC group. Multiple processes running from different UIC groups are synchronized.

All code paths in the image activator, the Install utility, and RMS that read the KFE list acquire PR mode locks on the KFE resource. Install utility code paths that read the KFE also acquire the Install lock in PR mode. To ensure that readers of the KFE list (particularly the image activator) are not blocked too long by multiple writers of the KFE (for example, several INSTALL ADD commands), Install utility code paths that write the KFE must acquire the Install lock in PW mode. Only the holder of a PW mode Install lock may attempt to convert the KFE lock to EX mode. Since only one writer at a time can acquire the PW mode Install lock, other writers queue for the Install lock rather than for the KFE lock.

When a writing process completes, it first converts the EX mode lock on the KFE to a PR mode lock. The only possible waiting requests are PR mode requests, and these are granted. The writer next dequeues the Install lock, allowing another writer to acquire it. This new writer requests an EX mode

lock on the KFE. The request is granted when the readers complete and release their PR mode locks.

The combination of these two locks guarantees that writers cannot block readers for extended time periods.

## A.8 NETACP LOCK USE

In VMScluster configurations, the network ancillary control process (NETACP) uses two categories of locks: locks to implement VMScluster alias functions and locks to implement network proxy access functions.

### A.8.1 VMScluster Alias Locks

#### A.8.1.1 Master Registration Lock

| | |
|---|---|
| Resource name string | "CLU$NETACP_" + alias node address |
| Symbol | None |
| Modes of acquisition | NL, CR, PW, EX |
| Domain | UIC-based domain |
| Access mode | Kernel |
| Parent | None |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | Owned by NETACP process |

The NETACP process on each VMScluster node that participates in the Alias Node service queues a lock on a resource whose name contains the alias node address. This lock is called the Master Registration lock (MRL) and is normally held at CR mode. The MRL is used as the parent lock for all other VMScluster alias locks.

The value block in the MRL contains a quadword bit mask, with a bit set for each VMScluster node participating in the VMScluster alias. A new node queues the MRL in PW mode, uses the value block to determine the first free alias index number from this bit mask, allocates that index number for its own use, updates the value block, and stores its alias index at NET$GW_CLUSTER_INDEX. The MRL is then converted to a NL mode lock, updating the value block. The new participant next converts the lock to EX mode. This forces delivery of blocking ASTs to the current members, notifying them of the new VMScluster alias member.

#### A.8.1.2 Individual Index Lock

| | |
|---|---|
| Resource name string | "IXL_" + alias index number |
| Symbol | None |
| Modes of acquisition | NL, EX |
| Domain | UIC-based domain |

| | |
|---|---|
| Access mode | Kernel |
| Parent | Master Registration lock |
| Value block | Yes |
| Blocking AST | None |
| Process context | Owned by NETACP process |

Each participant in the VMScluster alias scheme requires an Individual Index lock (IXL). An IXL is a sublock of the MRL and has a resource name formed from the string IXL_ + alias index number. Its value block contains registration data for the participating member, such as DECnet node address, alias maximum links, and routing/nonrouting status. Each new member queues an EX mode lock on its own IXL resource name. The member updates the value block with information about itself by lowering the lock to NL mode. Other participating members cycle through a set of lock states to allow each member to read the updated value block.

### A.8.1.3 Individual Departure Lock

| | |
|---|---|
| Resource name string | "IDL_" + alias index number |
| Symbol | None |
| Modes of acquisition | CR, EX |
| Domain | UIC-based domain |
| Access mode | Kernel |
| Parent | Master Registration lock |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by NETACP process |

Each participating VMScluster member queues an EX mode lock on its own Individual Departure lock (IDL) resource. All other members request a CR mode lock on that resource, with deadlock search disabled. If a CR request is ever granted, the other member knows that the original member that held the EX mode lock is no longer participating.

### A.8.1.4 Individual Link Registration Lock

| | |
|---|---|
| Resource name string | "ILR_" + alias index number |
| Symbol | None |
| Modes of acquisition | NL, CR, EX, PW |
| Domain | UIC-based domain |
| Access mode | Kernel |
| Parent | Master Registration lock |
| Value block | Yes |

| | |
|---|---|
| Blocking AST | Yes |
| Process context | Owned by NETACP process |

An Individual Link Registration (ILR) lock is a sublock of the MRL. Each alias member has an associated resource, whose name is the string ILR_ concatenated with the alias index number. A member always holds an ILR lock for itself in NL mode. In addition, a member that is a router holds a CR mode lock for itself and every other member. Each CR mode lock has an associated blocking AST.

An ILR lock is used for flow control between a participating alias node and one or more alias routers. When an alias member needs to communicate to each alias router that it can or cannot accept any more links, it converts its NL mode lock to PW mode. (Converting to PW mode instead of EX mode prevents premature triggering of the CR mode lock blocking AST on a router.) The PW mode lock is then converted to NL. As a result, the value block is updated with information about the current and maximum links for this member. Next, the NL mode lock is converted to EX mode to trigger the blocking AST on each router's CR mode lock for that member. Finally, the lock is converted back to NL mode.

When the blocking AST for the CR mode lock is triggered on each router, the CR mode lock is immediately converted to NL mode. Then each alias router requests an EX mode lock to read the value block. One of the routers will be granted the EX mode lock when all the routers have transitioned the lock from CR mode to NL mode. The router acquiring the EX mode lock then updates its tables based on the new value block, converts the lock to NL to allow another router to obtain the lock in EX mode, and finally converts the lock back to CR mode.

### A.8.2 Network Proxy Access Locks

The NETACP process uses standard RMS locks to synchronize access to the proxy file, NETPROXY.DAT. In addition, it uses the following locks to propagate the volatile proxy database changes to other VMScluster nodes.

#### A.8.2.1 Modified Proxy Lock

| | |
|---|---|
| Resource name string | "NET$NETPROXY_MODIFIED" |
| Symbol | None |
| Modes of acquisition | PR, PW |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | None |

| | |
|---|---|
| Blocking AST | Yes |
| Process context | Owned by NETACP process |

This is the main proxy lock, typically granted to all VMScluster nodes in PR mode with an associated blocking AST. If proxy information is modified on a participating node, the Authorize utility or the network management listener (NML) requests that NETACP obtain a new lock on this resource in PW mode. This triggers blocking AST delivery to the NETACP process on all VMScluster nodes, including the one that queues the PW lock, as notice of proxy modification.

### A.8.2.2 Proxy Function Lock

| | |
|---|---|
| Resource name string | "NET$NETPROXY_FNCT" |
| Symbol | None |
| Modes of acquisition | NL, EX |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | Yes |
| Blocking AST | None |
| Process context | Owned by NETACP process |

The Proxy Function lock is used to transmit the function to be performed on the NETPROXY.DAT file, for example, Rebuild_Proxy, Add_Proxy, and Delete_Proxy. The function code is transmitted in the value block. Holding this lock in EX mode also serializes NETACP's use of the Proxy Key locks.

### A.8.2.3 Proxy Key Locks

| | |
|---|---|
| Resource name string | "NET$NETPROXY_KEY" + key number |
| Symbol | None |
| Modes of acquisition | NL, EX |
| Domain | System domain |
| Access mode | Kernel |
| Parent | None |
| Value block | Yes |
| Blocking AST | None |
| Process context | Owned by NETACP process |

NETACP uses these locks and key values to determine whether a record with a specified key exists in the NETPROXY.DAT database used by the local node. The value blocks of these four key locks pass the RMS key values desired for the NETPROXY.DAT indexed file.

The four key numbers allow a total of 64 bytes of key information:

- NET$NETPROXY_KEY1 : first 16 bytes of the key
- NET$NETPROXY_KEY2 : second 16 bytes of the key
- NET$NETPROXY_KEY3 : third 16 bytes of the key
- NET$NETPROXY_KEY4 : fourth 16 bytes of the key

## A.9 JOB CONTROLLER AND QUEUE MANAGER LOCK USE

Since OpenVMS Alpha Version 1.5, queue manager functions have been separated from job controller functions. A new process called the queue manager performs the queue management functions previously performed by the job controller.

The job controller processes running on multiple VMScluster nodes and the queue manager process use a variety of locks to synchronize their activities. Many of these locks coordinate access to the queue file and the master file. Since the queue file and master file are accessed through RMS, standard RMS locking activity occurs as well (see Section A.6). This, however, is transparent to the job controller.

### A.9.1 New Master File Lock

| | |
|---|---|
| Resource name string | "QMAN$NEW_MASTER" |
| Symbol | None |
| Modes of acquisition | PR, EX |
| Domain | System domain |
| Access mode | User |
| Parent | None |
| Value block | None |
| Blocking AST | Yes |
| Process context | Owned by job controller process |

The New Master File lock is used to signal job controllers on other nodes of a VMScluster system that a new queue manager master file, QMAN$MASTER.DAT, has been created. A new master file is created in response to the DCL command START /QUEUE /MANAGER /NEW.

During startup each job controller in a VMScluster system requests a PR mode lock on the New Master File resource. To signal job controllers on other nodes of a VMScluster system that it has created a new master file, the job controller that created the new file requests the same lock in EX mode, triggering blocking ASTs on all other nodes. Upon receiving the blocking AST, each job controller opens the new master file. The signaling job controller dequeues the lock and requests it in PR mode, so that the blocking AST mechanism is rearmed.

### A.9.2 Master File Access Lock

| | |
|---|---|
| Resource name string | "QMAN$MSR_" + device name + file ID |
| Symbol | None |
| Modes of acquisition | NL, PR, PW |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Requested by job controller and queue manager processes |

The Master File Access lock synchronizes access to the queue manager master file. The queue manager process and all job controller processes in a VMScluster system request this lock in NL mode when they are started. Before reading the master file, the queue manager or job controller process requests a lock conversion to PR mode. Similarly, before writing to the master file, the queue manager or job controller process requests a lock conversion to PW mode.

This lock serves as the parent lock to many other job controller and queue manager locks.

### A.9.3 Master File Check Lock

| | |
|---|---|
| Resource name string | "JBC$_CHECK_DB" |
| Symbol | None |
| Modes of acquisition | PR, EX |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | Master File Access lock |
| Value block | None |
| Blocking AST | Yes |
| Process context | Requested by job controller and queue manager processes |

This lock is used to cause job controllers to check the queue manager master file for changes. Each job controller process normally holds a PR mode lock on the Master File Check resource.

When a job controller makes a change to the master file, it requests the lock in EX mode, triggering blocking ASTs on all other job controllers. The blocking AST causes each job controller to examine the master file records and take appropriate actions if necessary. Possible actions include starting

the queue manager, stopping the queue manager, or notifying waiting processes of the queue manager's termination.

After recording its startup status in the master file, the queue manager requests the lock in EX mode to force all job controllers to check the master file.

### A.9.4 Node Identifier Lock

| | |
|---|---|
| Resource name string | "JBC$" + node name |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by job controller process |

The job controller acquires the Node Identifier lock in EX mode during job controller startup before initiating communication with the queue manager. The lock ensures that only one job controller has access to the queue manager. While only one job controller exists during normal operation, a catastrophic failure could cause a situation in which a failing job controller is exiting as a new job controller starts. The Node Identifier lock protects the queue manager in such a case.

### A.9.5 New Job Controller Lock

| | |
|---|---|
| Resource name string | "QMAN$NEW_JOBCTL" |
| Symbol | None |
| Modes of acquisition | PR, EX |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | Master File Access lock |
| Value block | None |
| Blocking AST | Yes |
| Process context | Owned by queue manager process |

The New Job Controller lock helps the queue manager establish a link with each job controller in a VMScluster system. During its startup the queue manager requests a PR mode lock on this resource. When a node is added to the VMScluster, its job controller reads the master file's queue manager information, creates a queue manager data block for the node, and requests an EX mode lock on the New Job Controller resource. This triggers

a blocking AST on the queue manager, which establishes a communication link with the new node. Any process on the new node can then communicate with the queue manager using this communication link.

### A.9.6 Queue Manager Alive Lock

| | |
|---|---|
| Resource name string | "QMAN$JBC_ALIVE_" + queue manager ID |
| Symbol | None |
| Modes of acquisition | CR, PR, PW, EX |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | Master File Access lock |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by job controller process |

During startup, a job controller process requests a PR mode lock on the Queue Manager Alive resource using the LCK$V_NOQUEUE flag. If the request is not granted, the queue manager process has already been started (by another job controller in the VMScluster system); the job controller continues with other processing.

If the PR mode request is granted, the queue manager has not been started. Running on a VMScluster system, the job controller determines whether its node is the best candidate to run the queue manager process. If so, the job controller converts the PR mode lock to an EX mode lock and starts up the queue manager process. Otherwise, the job controller arms a timer, which upon expiration triggers it to attempt startup of the queue manager process anyway.

During its startup the queue manager process queues a request for the Queue Manager Alive lock in CR mode. Since the job controller already holds the lock in EX mode, the request will be queued. If the CR mode request is ever satisfied, the queue manager process will exit. The queue manager's CR mode lock is granted only when the job controller releases the EX mode lock, which happens only when the job controller decides to terminate the queue manager or the job controller itself terminates.

### A.9.7 Queue Definition and Journal Files Lock

| | |
|---|---|
| Resource name string | "QMAN$OWNER_" + database number |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | UIC-based domain |
| Access mode | User |

| | |
|---|---|
| Parent | Master File Access lock |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by queue manager process |

When a queue manager starts, it requests an EX mode lock on the Queue Definition and Journal Files resource. The resource name includes a database number, or queue file identifier, indicating which QMAN$QUEUE and QNAM$JOURNAL files the queue manager is associated with. If the lock request is denied, indicating that another queue manager process has already started for this database, the second queue manager process exits. Only one queue manager can exist for a given database on a VMScluster system running OpenVMS Alpha Version 7.0.

### A.9.8 Forms Cache Lock

| | |
|---|---|
| Resource name string | "QMAN$REF" + master record ID |
| Symbol | None |
| Modes of acquisition | CR, PR, PW, EX |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | Master File Access lock |
| Value block | None |
| Blocking AST | Yes |
| Process context | Owned by queue manager process |

Forms and characteristics can be shared by many queue managers. Each maintains a cache of its own active forms and characteristics, and must therefore be notified of changes and have the opportunity to block deletion of information, for instance, deletion of a form currently in use.

A queue manager acquires a PR mode Forms Cache lock with an associated blocking AST routine when it places an object into service and therefore into its cache. Before modifying or deleting an object, the queue manager converts the lock to PW mode, forcing blocking ASTs to fire for any other queue manager holding a cached copy. The blocking AST routine marks the cached data invalid and either dequeues its lock, if no queues or jobs currently reference the object, or rearms the lock.

At this point, only queue managers with active references to the object hold Forms Cache locks on it. If this is an attempt to delete the object, the deleting queue manager requests an EX mode lock conversion with the LCK$V_NOQUEUE flag set. If the conversion completes successfully, no references exist for the form or characteristic and it can be deleted; otherwise the request is denied.

### A.9.9 Queue ORB Lock

| | |
|---|---|
| Resource name string | "ORB$JBC_" + queue ID |
| Symbol | None |
| Mode of acquisition | PW |
| Domain | System domain |
| Access mode | User |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by process changing access control on the queue |

When the access control list on a queue is being modified, for example, with the DCL command EDIT/ACL, a PW mode lock on the Queue ORB resource is requested on behalf of the process modifying the list. This lock ensures that the queue manager or any other process does not modify the ORB for a queue while the queue's access control list is being changed by another process.

### A.9.10 Master Queue Lock

| | |
|---|---|
| Resource name string | "QMAN$MASTER_QUEUES" |
| Symbol | None |
| Modes of acquisition | NL, PR, PW |
| Domain | UIC-based domain |
| Access mode | User |
| Parent | Master File Access lock |
| Value block | Yes |
| Blocking AST | Yes |
| Process context | Requested by job controller and queue manager processes |

The Master Queue lock is normally held in PR mode by a queue manager. When a new queue is added or an existing queue is deleted, the job controller uses this lock to notify the queue manager of the change. The job controller requests a NL mode lock on the resource and converts it to PW, causing blocking ASTs to fire. The blocking AST routine, in the context of the queue manager process, converts the PR mode lock to NL, allowing the PW mode request to be granted, and queues another PR mode request to rearm the blocking AST. The job controller fills in the value block with the ID of the changed queue and dequeues the PW mode lock. The queue manager's PR mode request is granted, and it obtains the changed queue ID from the value block.

## A.10 SYSMAN LOCK USE

### A.10.1 Driver-Loading and Autoconfiguration Lock

| | |
|---|---|
| Resource name string | CSID + "IOGEN$LOCK" |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | User |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by driver-loading process |

The SYSMAN IO commands LOAD, RELOAD, AUTOCONFIGURE, and CONNECT require exclusive access to a system's SYSMAN I/O database. SYSMAN protects its database from concurrent access by SYSMAN executing in multiple processes simultaneously through an EX mode lock on the Driver-Loading and Autoconfiguration resource.

### A.10.2 SMISERVER Main Lock

| | |
|---|---|
| Resource name string | "SMISERVER$" + SCSNODE |
| Symbol | None |
| Mode of acquisition | EX |
| Domain | System domain |
| Access mode | Executive |
| Parent | None |
| Value block | None |
| Blocking AST | None |
| Process context | Owned by SMISERVER process |

The SMISERVER process uses the Main lock to ensure that only one SMISERVER process at a time exists on a system. It requests an EX mode lock on the resource using the LCK$V_NOQUEUE flag. If the request is not granted immediately, another SMISERVER process is already running; the current one is redundant and deletes itself.

When SMISERVER performs a clusterwide Set Time ($SETIME) system service, it synchronizes using the Set Time lock (see Section A.2.2).

# Selected Acronyms

These acronyms are selected from those that appear in this book. This list is not exhaustive; for instance, acronyms for facilities, programs, and instructions are not included.

| *Acronym* | *Meaning* |
|---|---|
| ACB | AST control block |
| ACL | access control list |
| ACP | ancillary control process |
| ADP | adapter control block |
| AI | argument information (register) |
| AIB | ACP I/O buffer |
| AP | argument pointer (register) |
| APLD | activated privileged library dispatch vector entry |
| AQB | ACP queue block |
| ARB | access rights block |
| ARG | bootstrap argument list |
| ASCII | American Standard Code for Information Interchange |
| ASN | address space number |
| AST | asynchronous system trap |
| ASTEN | AST enable (register) |
| ASTSR | AST summary register |
| BDB | buffer descriptor block |
| BDTAB | boot driver table structure |
| BLB | buffer lock block |
| BOD | buffer object descriptor |
| BOOPAR | boot parameter argument list |
| BOOTCB | boot control block |
| BRK | breakthrough message descriptor block |
| BUSARRAY | adapter bus array |
| CCB | channel control block |
| CD-ROM | compact disk read-only memory |
| CDDB | class driver data block |
| CDRP | class driver request packet |
| CEB | common event block |

| *Acronym* | *Meaning* |
|---|---|
| CEF | common event flag wait (scheduling state) |
| CFCB | cache file control block |
| CI | computer interconnect |
| CLI | command language interpreter |
| CMDTABLE | adapter command table |
| COLPG | collided page wait (scheduling state) |
| COM | computable (scheduling state) |
| COMO | computable outswapped (scheduling state) |
| CPU | central processing unit |
| CPUTAB | CPU table structure |
| CRAB | counted resource allocation block |
| CRAM | controller register access mailbox |
| CRB | controller request block |
| CRCTX | counted resource context block |
| CRD | corrected read data |
| CSB | cluster system block |
| CSID | cluster system ID |
| CSR | control/status register |
| CUR | currently executing (scheduling state) |
| CWPS | clusterwide process services |
| CWPSSRV | clusterwide process services data block |
| DAT | data alignment trap |
| DATFX | data alignment trap fixup (register) |
| DCL | Digital command language |
| DDB | device data block |
| DDIF | Digital Document Interchange Format |
| DDT | driver dispatch table |
| DMA | direct memory access |
| DPT | driver prologue table |
| DSSI | Digital Storage Systems Interconnect |
| DTB | data stream translation buffer |
| ECC | error correction code |
| EIHD | OpenVMS Alpha image header |
| EIHI | image ident area |
| EISD | image section descriptor |
| EPID | extended process ID |

| *Acronym* | *Meaning* |
|---|---|
| ER | executive read (protection) |
| ERKW | executive read, kernel write (protection) |
| ESP | executive mode stack pointer (register) |
| ESR | exception service routine |
| EW | executive write (protection) |
| FAB | file access block |
| FCB | file control block |
| FDDI | Fiber Distributed Data Interconnect |
| FDT | function decision table |
| FEN | floating-point enable (register) |
| FID | file identification number |
| FIFO | first-in/first-out |
| FKB | fork block |
| FP | frame pointer (register) |
| FPCR | floating-point control register |
| FPG | free page wait (scheduling state) |
| FRED | floating-point register and execution data structure |
| GB | gigabyte |
| GBD | global buffer descriptor |
| GBH | global buffer header |
| GBS | global buffer section |
| GBSB | global buffer synchronization block |
| GPT | global page table |
| GPTE | global page table entry |
| GPTX | global page table index |
| GQC | Get Queue Information system service context block |
| GSD | global section descriptor |
| GST | global section table |
| GSTE | global section table entry |
| GSTX | global section table index |
| HIB | hibernate wait (scheduling state) |
| HIBO | hibernate wait (outswapped) (scheduling state) |
| HWPCB | hardware privileged context block |
| HWRPB | hardware restart parameter block |
| HWRPB_CRB | console routine block |

| *Acronym* | *Meaning* |
|---|---|
| HWRPB_CTB | console terminal block |
| HWRPB_PMD | physical memory descriptor |
| HWRPB_PMR | physical memory region descriptor |
| I/O | input/output |
| ID | identification |
| IDB | interrupt dispatch block |
| IFAB | internal file access block |
| IFI | internal file identifier |
| IHD | OpenVMS VAX image header |
| IMCB | image control block |
| IOSB | I/O status block |
| IOVEC | boot device I/O vector |
| IPC | interprocess communication |
| IPID | internal process ID |
| IPIR | interprocessor interrupt request (register) |
| IPL | interrupt priority level |
| IRAB | internal record access block |
| IRP | I/O request packet |
| IRPE | IRP extension |
| ISR | interrupt service routine |
| ITB | instruction stream translation buffer |
| JIB | job information block |
| KB | kilobyte |
| KFD | known file directory |
| KFE | known file entry |
| KFERES | known file entry resident section descriptor |
| KFPB | known file pointer block |
| KFRH | known file resident image header |
| KPB | kernel process block |
| KR | kernel read (protection) |
| KRP | kernel request packet |
| KSP | kernel mode stack pointer (register) |
| KTB | kernel thread block |
| KW | kernel write (protection) |
| L1PT | level 1 page table |

| *Acronym* | *Meaning* |
|---|---|
| L1PTE | level 1 page table entry |
| L2PT | level 2 page table |
| L2PTE | level 2 page table entry |
| L3PT | level 3 page table |
| L3PTE | level 3 page table entry |
| LAT | local area transport |
| LBN | logical block number |
| LCKCTX | lock context block |
| LDRHP | loader huge page descriptor |
| LDRIMG | loader image data block |
| LEF | local event flag wait (scheduling state) |
| LEFO | local event flag wait (outswapped) (scheduling state) |
| LIFO | last-in/first-out |
| LKB | lock block |
| LKSB | lock status block |
| LMB | logical memory block |
| LNMB | logical name block |
| LNMC | logical name table name cache block |
| LNMHSH | logical name hash table |
| LNMTH | logical name table header |
| LNMX | logical name translation block |
| MB | megabyte |
| MBPR | mailbox pointer register |
| MCES | machine check error summary (register) |
| MSCP | Mass Storage Control Protocol |
| MWAIT | miscellaneous wait (scheduling state) |
| NA | no access (protection) |
| NI | network interconnect |
| ODS-2 | On-Disk Structure Level 2 |
| ORB | object rights block |
| P0PT | P0 page table |
| P0PTE | P0 page table entry |
| P1PT | P1 page table |
| P1PTE | P1 page table entry |
| PA | physical address |

| *Acronym* | *Meaning* |
|---|---|
| PALcode | privileged architecture library code |
| PC | program counter (register) |
| PCB | process control block |
| PCBB | privileged context block base (register) |
| PCC | process cycle counter |
| PDT | port descriptor table |
| PFL | page file control block |
| PFLMAP | page/swap file mapping window block |
| PFN | page frame number |
| PFW | page fault wait (scheduling state) |
| PHD | process header |
| PID | process identifier |
| PKTA | per kernel thread area |
| PLV | privileged library vector |
| PME | performance monitoring enable (register) |
| PMI | processor-memory interconnect |
| POSIX | Portable Operating System Interface |
| PPD | process-permanent data |
| PQB | process quota block |
| PRBR | processor base register |
| PS | processor status (register) |
| PSCANCTX | $PROCESS_SCAN context block |
| PSECT | program section |
| PST | process section table |
| PSTE | process section table entry |
| PSTX | process section table index |
| PSXFR | resource wait—POSIX fork creation (scheduling state) |
| PT | page table |
| PTBR | page table base register |
| PTE | page table entry |
| PV | procedure value (register) |
| RA | return address (register) |
| RAB | record access block |
| RAM | random access memory |
| RCB | routing control block |
| RDAB | resource domain access block |

| *Acronym* | *Meaning* |
|---|---|
| RDDB | resource domain data block |
| RDE | region descriptor entry |
| RDPB | resource domain pointer block |
| RFA | record file address |
| RISC | reduced instruction set computer |
| RLB | record lock block |
| RMS | Record Management Services |
| RSB | resource block |
| RU | recovery unit |
| RVT | relative volume table |
| RWAST | resource wait—AST (scheduling state) |
| RWCAP | resource wait—CPU capability (scheduling state) |
| RWCLU | resource wait—VMScluster transition (scheduling state) |
| RWCSV | resource wait—VMScluster server (scheduling state) |
| RWMBX | resource wait—mailbox full (scheduling state) |
| RWMPB | resource wait—modified page writer busy (scheduling state) |
| RWMPE | resource wait—modified page list empty (scheduling state) |
| RWNPG | resource wait—nonpaged dynamic memory (scheduling state) |
| RWPAG | resource wait—paged dynamic memory (scheduling state) |
| RWPFF | resource wait—page file full (scheduling state) |
| RWSCS | resource wait—distributed lock manager (scheduling state) |
| SB | system block |
| SCB | system control block |
| SCBB | system control block base (register) |
| SCC | system cycle counter |
| SCS | system communication services |
| SCSI | Small Computer System Interconnect |
| SFSB | shared file synchronization block |
| SHL | shareable image list entry |
| SI | storage interconnect |
| SIRR | software interrupt request register |
| SISR | software interrupt summary register |
| SLOT | per-CPU slot |
| SMP | symmetric multiprocessing |
| SP | stack pointer (register) |
| SPL | spinlock control block |

| *Acronym* | *Meaning* |
|---|---|
| SPT | system page table |
| SPTE | system page table entry |
| SR | supervisor read (protection) |
| SREW | supervisor read, executive write (protection) |
| SRKW | supervisor read, kernel write (protection) |
| SSP | supervisor mode stack pointer (register) |
| SUSP | suspend wait (scheduling state) |
| SUSPO | suspend wait (outswapped) (scheduling state) |
| SW | supervisor write (protection) |
| SWRPB | software restart parameter block |
| SYSAP | system application |
| TAST | terminal AST block |
| TB | terabyte, translation buffer |
| TBCHK | translation buffer check (register) |
| TEB | thread environment block |
| TMSCP | tape MSCP |
| TQE | timer queue entry |
| UAF | user authorization file |
| UCB | unit control block |
| UIC | user identification code |
| UR | user read (protection) |
| UREW | user read, executive write (protection) |
| URKW | user read, kernel write (protection) |
| URSW | user read, supervisor write (protection) |
| USP | user mode stack pointer (register) |
| UW | user write (protection) |
| VA | virtual address |
| VBN | virtual block number |
| VCB | volume control block |
| VEC | interrupt transfer vector |
| VLE | vector list extension |
| VMS | virtual memory system |
| VPN | virtual page number |
| VPTB | virtual page table base (register) |
| WCB | window control block |

| *Acronym* | *Meaning* |
|---|---|
| WHAMI | who am I (register) |
| WQH | wait queue header |
| WSLE | working set list entry |
| WSLX | working set list index |
| XMI | Extended Memory Interconnect |
| XQP | Extended QIO Processor |

# Index


**ACB (AST control block)**. *See also* LKB; TQE.
field indicating thread-safe AST procedure, 185
fields related to multithreading, 196–197, 196 (fig.)
initialized by
EXE$CREATE_KERNEL_THREAD, 171
EXE$DELPRC, 420
EXE$REGISTER_THREAD_MANAGER, 166
LKB component, LKB fields that specify, 118
LKB use as, 135
marked as thread-safe, 239
removed from PCB queues during image rundown, 376
use by EXE$REGISTER_THREAD_MANAGER, 166
**ACB$V_THREAD_SAFE bit (ACB$L_FLAGS field)**
definition and use, 185
**access control**. *See* access protection.
**access control list**. *See* ACL.
**access modes**. *See also* protection.
changing to less privileged, 295
$EXIT operations affected by, 366
images, IMCB field that specifies, 317
inner. *See* protection.
outer. *See* protection.
resources
how determined, 110, 111
lock information access restricted by, 139
$RUNDWN operations affected by, 370
suspended kernel thread, SUSP and SUSPO categorized by, 35
system-owned locks, how determined, 136
when accessing resource domain, 128, 129
when calling rundown routines, 370
**access protection**. *See* ARB; protection; UIC.
**access rights block**. *See* ARB.
**access violations**
fault-on-execute ESR emulation of, 413
**accessor**
term definition, 485
**ACCOUNT module**. *See* EXE$PRCDELMSG routine.
**ACL (access control list)**. *See* ORB.
**ACP (ancillary control process)**. *See* AQB; I/O; XQP.
**ACP$BADBLOCK_MBX mailbox**. *See* mailboxes.
**activated privileged library dispatch vector**
declared rundown routine entries, 377
fields related to rundown routines, 378 (fig.)
reset during image activation, 326
zeroed by DELETE, 422
**activating**
command language interpreters, 345
images, term definition, 299
known images, 345
shareable images, 335–338
example, 343–344
simple main images, 326–335
translated images, 347
**.ADDRESS directive**
characteristics and modification during image activation, 348
**address fixup**. *See also* fixup section; $IMGFIX.
code, 355 (fig.)
$IMGFIX operations, 355
described by subset fixup section, 349
effect on linkage section, 306
EXE$IMGFIX operations, 359–360
image section targeted by, 310
Install utility actions, 310
linkage pair, 355 (fig.)
$IMGFIX operations, 356
linkage pair with procedure signature blocks, 357 (fig.)
$IMGFIX operations, 356–357
longword and quadword, 354 (fig.)
$IMGFIX operations, 354
optimizing, 358–359
overview, 348–350
performed by image activator, 349
processing
at image activation, 358
at image installation, 358
term definition, 300, 348
**address references**
inter-image, adjustment, 348
intra-image, must be relocated, 348, 349
requiring adjustment, 348–349
code, 348
linkage pair, 348
linkage pair with procedure signature blocks, 349
longword, 348
quadword, 348
**address relocation**
described by subset fixup section, 349
effect on linkage section, 306
image activation, $IMGFIX operations, 353
information contained in fixup section, 310
Install utility actions, 310

**address relocation** *(continued)*
optimizing, 358–359
overview, 348–350
performed by image activator, 349
processing
at image activation, 358
at image installation, 358
records
characteristics and use, 352–353
layout, 353 (fig.)
term definition, 300, 348
**Address Relocation Fixup system service**. *See* $IMGFIX.
**address space**. *See* memory management; pages; protection.
P0. *See* P0 space.
P1. *See* P1 space.
**address space match bit (PTE)**
effect on TB invalidation, 24
**address space number**. *See* ASN.
**address translation**. *Consult* Alpha Architecture Reference Manual.
use of TB, 24
**Affect Process Scheduling system service**. *See* $SCHED.
**affinity**. *See* CPU affinity.
**AFFINITY_SKIP parameter (SYSGEN)**
initializing KTB$L_AFFINITY_SKIP field with, 45
**AI (argument information) register**
characteristics and use, 396
**ALLSPOOL (allocate spooled device privilege)**. *See* privileges.
**Alpha architecture**. *See also* access modes; exceptions; HWPCB; memory management; PTE; TB.
architectural support for translated images, 389–390
atomic memory access, characteristics, 388–389
compared to VAX, 386
memory references, compared to VAX, 387–389
precision exception handling, compared to VAX, 387
**ALTPRI (set any priority value privilege)**. *See also* privileges.
effect on kernel thread adjusting its priority, 28
required for
implicit affinity modification, 254
kernel thread priority modification, 28
process affinity modification, 254
process capabilities modification, 255
process priority modification, 255
use by EXE$CREPRC, 268, 292
use during kernel thread creation, 171
**AMAC$EMUL_CALL procedure**
function, 400
**AMOVRM routine (PALcode)**
operations, 388, 390
**AMOVRR routine (PALcode)**
operations, 390
**ancillary control process**. *See* ACP.
**AP (argument pointer) register**
characteristics and use, 396
**APLD$L_EXEC_RUNDOWN_COUNT field**
number of image-temporary executive mode rundown routines, 377
**APLD$L_EXEC_RUNDOWN_PERM field**
number of permanent executive mode rundown routines, 377
**APLD$PS_KERN_RUNDOWN_VECTOR field**
offset to kernel mode rundown routine list, 378
**ARB (access rights block)**. *See also* protection.
initializing during process creation, 270
location, 271
obtaining information from, $GETJPI operations, 238
**ARB$Q_PRIV field**. *See also* privileges.
process privilege mask, use and routines that manipulate, 380 (table)
use by LOGINOUT, 447
**argument information register**. *See* AI register.
**argument pointer register**. *See* AP register.
**arguments**
formal name, xiv
passing to a procedure, Alpha compared to VAX, 396
**arithmetic exceptions**. *See* condition handlers; exceptions.
**$ASCEFC (Associate Common Event Flag Cluster system service)**
creating common event flags, 84, 88
**.ASCID directive**
characteristics and modification during image activation, 348
**ASN (address space number)**
characteristics and use, 24–26
effect on TB invalidation, 25
FRED field that contains, 25
PCB field that contains, 25
per-CPU database field that contains, 25
register, as part of hardware context, 20
rescheduling interrupt service routine use of, 73–74
reusing, 26–27
sequence number
FRED field that contains, 25
PCB field that contains, 25
per-CPU database field that contains, 25
rescheduling interrupt service routine use of, 73
**Associate Common Event Flag Cluster system service**. *See* $ASCEFC.

**associated mailboxes**. *See* mailboxes.
**AST (asynchronous system trap)**. *See also* ASTEN register; ASTSR; attention ASTs; interrupts; PKAST; SCH$ASTDEL routine; SCH$QAST routine; special kernel mode ASTs; synchronization; *Consult* OpenVMS Programming Concepts.
addresses of completion and blocking procedures, LKB field that contains, 119
blocking
system-owned lock cannot include, 472
while a process is suspended, 251
completion
system-owned lock cannot include, 472
CWPS$SRCV_GETJPI_AST routine, operations, 240
data structures required for multithreading, 196–197
DELETE kernel mode AST procedure, control flow, 421–426
delivery. *See* AST delivery.
$FORCEX use of, 252
JPI$V_NO_TARGET_AST, flag definition and use, 238
kernel mode
$DELETE_KERNEL_THREAD use of, 175
$END_MULTITHREAD use of, 167
process deletion use of, 421–426
use during deletion of single-image process, 438
kernel thread queues
servicing, 199, 201
out-of-band
enabled by DCL, 454
process queues
servicing, 199, 200
process suspension use of, 246
queues
initialized by EXE$CREPRC, 274
in multithreaded process, 197
summary bits, 197
queuing
after granting a lock, requirements for, 134
during process suspension, 247
effect on kernel threads in event flag wait state, 98
effect on mutex-waiting kernel thread, 40
effect on wait states, 35
event causing kernel thread state change, 60
by EXE$GETJPI, 239
linking LKB into lock timeout queue with LKB AST queue field, 142 (fig.)
makes the kernel thread computable, 37
to a multithreaded process, 197–198
in remote $GETJPI request, 241
SUSP and SUSPO state effect dependent on access mode, 35
to a suspended process, 251
to a target process during process deletion, 421
transitions caused by, 35, 36
quota
shared by kernel threads in a process, 237
termination procedure, activation by CTRL/Y processing, 459
thread-safe process deletion, 420
user mode
characteristics and use, 218–220
tagged, actions taken, 220
untagged, actions taken, 218
wait state. *See* resource wait—RSN$_ASTWAIT.
**AST delivery**
during kernel thread wait, mechanism that permits, 52, 93, 97, 103
enabling
during kernel thread deletion, 176
inner mode
in a multithreaded process, 186, 199
during kernel thread deletion, 176
inner mode semaphore use during, 201
interrupt service routine. *See* AST delivery interrupts.
to kernel thread
in HIB, 52
in MWAIT, 54
in PFW, FPG, or COLPG, 53
requesting a resource lock, 108
in SUSP, 54
in wait state, 52
in a multithreaded process, 195, 199–201
process deletion use of, 417
system-owned lock queue restrictions resulting from, 136
**AST delivery interrupts (IPL 2)**. *See also* SCH$ASTDEL routine.
simulated by SCH$INTERRUPT, 74
**AST enable register**. *See* ASTEN register.
**AST summary register**. *See* ASTSR.
**ASTDEL module**
SCH$ASTDEL. *See* SCH$ASTDEL routine.
SCH$ASTDEL_K. *See* SCH$ASTDEL_K routine.
SCH$QAST. *See* SCH$QAST routine.
**ASTEN (AST enable) register**
as part of hardware context, 3, 20
**ASTSR (AST summary register)**
as part of hardware context, 3, 20
**ASTWAIT resource**. *See* resource wait—RSN$_ASTWAIT.
**asynchronous lock request completion**
queuing special kernel mode AST as a result of, 135
**asynchronous system trap**. *See* AST.

**ATTACH command**
operations, 465–466
**attention ASTs.** *See also* AST.
CTRL/Y enabled by DCL, 454
**audit server.** *See* AUDIT_SERVER process.
**AUDIT_SERVER process**
RDDB use by, 115
**AUTOGEN.COM command file.** *Consult* OpenVMS System Manager's Manual.
**automatic working set limit adjustment.** *See also* working set list.
disabling, 182

**backing store.** *See* memory management.
**bad blocks.** *See* disks; errors.
**BADPRCPGFLC bugcheck.** *See* bugchecks.
**BADQHDR bugcheck.** *See* bugchecks.
**BADVECTOR bugcheck.** *See* bugchecks.
**bag routine**
characteristics and use, 394
executed before translated executable image, 404
**balance set slots.** *See* memory management; PHD.
**base priority.** *See* priorities—base.
**batch processes.** *See also* job controller.
creating, 442 (fig.)
arguments passed to $CREPRC that result in, 441 (table)
by job controller process, 439
queue manager requests, 439
as result of SUBMIT command, 441
job step initialization, LOGINOUT control flow, 469–470
LOGINOUT image operations, control flow, 451–452
termination, 470
**binary translation**
advantages, 383
design goals, 383–384
**bit fields**
how represented, xiv
**BLI$CALLG procedure**
function, 400
**blocking ASTs.** *See* AST; locks.
**blocks.** *See* bad blocks.
**BOOT_HANDLER condition handler**
functions and operations, 363–364
**bound procedures**
characteristics and use, 398
intermediate procedures for nonminimal jacketing, 400
**bridge procedure**
characteristics and use, 398
**Bucket lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 489–491

**buffered I/O**
byte count quota, transition states triggered by, 41
**BUGCHECKFATAL parameter (SYSGEN).** *See* bugchecks.
**bugchecks.** *See also* errors.
fatal user mode, generated by EXE$KTHREAD_START, 173
FILCNTNONZ, generated by DELETE, 423
INCON_SCHED
generated by EXE$CALLBACK_JACKET, 208
generated by EXE$INFORM_TM_CALL, 207
generated by SCH$FIND_NEXT_PROC, 72
INCON_SHELL
generated by SWP$SHELL_INIT, 288
LOCKMGRERR, generated by the executive, 122
NOCALLTRANS
fatal, generated by EXE$NATIVE_TO_TRANSLATED, 402
nonfatal, generated by EXE$NATIVE_TO_TRANSLATED, 402
**BUGCHK (make bugcheck log entries privilege).** *See* bugchecks; privileges.
**BYPASS (bypass all object access controls privilege).** *See* privileges.
**byte count quota**
charged
by EXE$GETJPI, 238
waiting for, context, 54
**byte index**
term definition, xiv
**BYTLM (Authorize Utility quota).** *See* byte count quota.

**cache.** *See* memory management; TB.
**Cache lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 482–485
**call frames**
encapsulated VAX
characteristics and use, 410
example, 411 (fig.)
**calling standard.** *See also* condition handlers; procedures.
features that support interoperability, 397
OpenVMS Alpha compared to VAX, 396
translated images compared to native images, 396
**Cancel Exit Handler system service.** *See* $CANEXH.
**$CANEXH (Cancel Exit Handler system service)**
operations, 365
**canonical kernel stack**
arrangement, 23 (fig.)
characteristics and use, 23–24
for newly created kernel thread, 172

routines that complete, 50–51, 73
**capabilities**. *See* CPU capabilities.
**CAPABILITY module**. *See* SCH$ACQUIRE_AFFINITY routine; SCH$ADD_CPU_CAP routine; SCH$CLEAR_CAPABILITY routine; SCH$REMOVE_CPU_CAP routine; SCH$SET_AFFINITY routine; SCH$SET_CAPABILITY routine.
**catch-all condition handlers**
establishing
as oldest user mode frame-based handler, 296
as user mode last chance handler during process creation, 296
extensions
calling, during image execution, 363
**CCB$V_IMGTMP bit (CCB$B_STS field)**
tested by EXE$KRNDWN, 373
**CEB (common event block)**
address for clusters 2 and 3, 85
CEF wait queue use, 16, 85 (fig.)
field definitions, 84–86
layout, 84 (fig.)
linking into systemwide list, 85
naming, 85
status field definition and use, 85
**CEB mutex**. *See also* EXE$GL_CEBMTX cell.
name, 85
use by
EXE$ASCEFC, 88
EXE$DACEFC, 89
EXE$DLCEFC, 90
**CEF (common event flag wait state)**. *See also* event flag wait state; event flags; kernel thread states; wait states.
characteristics and use, 98
context for, 53
kernel threads, queue listheads, 16, 85 (fig.), 86
placing kernel threads into, 93, 97
removing kernel threads from, 101
transitions
from CEF to COM or COMO, 35
to CEF from CUR, 35
**central processing unit**. *See* CPU.
**change mode**. *See* exceptions; system services.
**Change to Executive Mode system service**. *See* $CMEXEC.
**Change to Kernel Mode system service**. *See* $CMKRNL.
**change-mode-to-supervisor handler**
declared by CLI, 454
**channels**
deassigning
during image rundown, 372
**CHME exception**. *See* change mode.
**CHMK exception**. *See* change mode.
**CHMS exception**. *See* change-mode-to-supervisor handler.
**class scheduling**
characteristics and use, 78–80
data structures affected by, 78
triggered at quantum end, 57
**classes**. *See* scheduling—classes.
**CLASS_SCHEDULER executive image**
characteristics and use, 78
**Clear AST system service**. *See* $CLRAST.
**Clear Event Flag system service**. *See* $CLREF.
**CLI (command language interpreter)**
callback mechanism, address of service routine, 453
command processing loop
commands handled by internal procedures, 456 (table)
control flow, 455 (fig.)
operations, 454 (table)
command table name copied at process creation, 273
exit handler, control flow, 458–459
image activation of, 7, 345, 448
image initiation
argument list passed to image, 458
operations, 457–458
image processing and, 452–468
initialization, operations, 453–454
name copied at process creation, 273
processes that map, control flow, 467 (fig.)
role in multi-image process, 434
**CLIS_INVREQTYP error status**
returned when no CLI exists, 295
**CLISYMTBL parameter (SYSGEN)**
size of CLI symbol table, use by LOGINOUT, 449
**clocks**. *See* timers.
**$CLRAST (Clear AST system service)**
CTRL/Y processing, operations, 459
**$CLREF (Clear Event Flag system service)**
clearing event flags, operations, 102
**cluster**. *See* common event flags—clusters; VMScluster systems.
**CLUSTER linker option**
effect on pagelet fault cluster factor for sections, 316
**cluster system ID**. *See* CSID.
**clusterwide process services**. *See* CWPS.
**clusterwide process services data block**. *See* CWPSSRV.
**CLUSTER_SERVER process**
clusterwide process services, 228
creating remote process, 269
deleting remote process, 424
forcing remote process to exit, 253
$GETJPI

**CLUSTER_SERVER process** *(continued)*
$GETJPI *(continued)*
special kernel mode AST operations in, 241–242
use by, 240
queuing kernel mode AST to, 228
relation to RWCSV wait, 39
resuming remote process, 252
setting priorities for remote process, 255
suspending remote process, 247
**CMEXEC (change mode to executive privilege).** *See* privileges.
**CMKRNL (change mode to kernel privilege).** *See also* privileges.
use by EXE$CREPRC, 268, 277, 279
**code flow**
figures that describe, xv
**collided page wait state.** *See* COLPG.
**COLPG (collided page wait state).** *See also* kernel thread states; page faults; wait states.
context for, 53
kernel thread formerly in PFW state, 36
transitions
from COLPG to COM or COMO, 37
to COLPG from CUR, 37
**COM (computable state).** *See also* COMO; kernel thread states; wait states.
characteristics, 34–35
kernel threads, queue listheads, location, 16 (fig.)
transitions
to COM from COLPG , 37
to COM from COMO, 290
to COM from CUR, 32
to COM from FPG, 37
to COM from HIB, 35
to COM from LEF or CEF, 101
from COM to other states, 34
to COM from PFW, 36
to COM from wait states, 35
**command language interpreter.** *See* CLI.
**common event block.** *See* CEB.
**common event flag wait state.** *See* CEF.
**common event flags.** *See also* CEB; CEF.
characteristics and use, 84–86
clusters
2 and 3, location, 85
associate access, 88
associating to, 85, 88–89
changing permanent to temporary, 90
characteristics, 81
deleting permanent, 90
dissociating from, 89–90
identifying, 85
processes currently associated to, 86
wait queues, 86
interprocess communication mechanism, 259
interprocess synchronization through, 104
load-locked/store-conditional mechanism use of, 86
location, 86
numbers available to processes, 81
SCH$POSTEF actions when setting, 101
sharing, 85
TQE quota use, 85
wait queues, 85 (fig.)
**communication mailbox.** *See* mailboxes—communication.
**communications.** *See also* common event flags; global sections; locks; logical names; mailboxes.
interprocess, mechanisms, 259–261
processes, list of available mechanisms, 223
**COMO (computable outswapped state).** *See also* COM; kernel thread states; wait states.
characteristics, 34
initial kernel thread state, 34
kernel threads, queue listheads, location, 16 (fig.)
state of newly created process, 263
transitions
to COMO from COLPG , 37
to COMO from COM, 34
to COMO from FPG, 37
to COMO from HIBO, 35
to COMO from LEFO or CEF, 101
from COMO to other states, 34
to COMO from PFW, 36
to COMO from SUSPO, 36
to COMO from wait states, 35
**compatible locks.** *See* locks.
**completion AST.** *See* AST; I/O; locks.
**compressed data sections**
term definition, 307
**computability**
system services affecting, 243–253
**computable outswapped state.** *See* COMO.
**computable state.** *See* COM.
**concurrency.** *See* locks; SMP systems; VMScluster systems.
**condition codes**
translated image use of, 385
**condition handlers.** *See also* AST; bugchecks; catch-all condition handlers; conditions; exceptions; image activation; last chance condition handlers; process creation; *Consult* OpenVMS Programming Concepts Manual.
last chance
multiple-image process, 466
traceback handler
functions, 363
operations, 363–364

use by EXE$IMGSTA in single-image process, 436
**context**. *See* hardware context; HWPCB; process context; system context.
**CONTINUE command**
operations, 460
**Contract Region system service**. *See* $CNTREG.
**control flags**
$GETJPI, AST queuing control provided by, 238
**control flow**. *See* exceptions; interrupts; scheduling; wait states.
**control mechanisms**. *See* AST; change mode—dispatchers; condition handlers; exceptions; interrupts; software interrupts.
**control region**. *See also* address space.
obtaining information from, $GETJPI operations, 234
process, retrieving information from, 238
swapper process does not contain, 238
**conventions**
figures, xiv–xv
text, xii–xiv
**conversion deadlocks**
detecting, method for, 143
searching for, preventing locks from participating, 142
victim selection in, 148–149
**conversion grant mode**. *See also* locks.
characteristics and use, 132
**conversion queue**
removing LKBs from, 139
resource, listhead location in RSB, 112
**copy-on-reference pages**
linkage section created as, 310
**CPB$V_IMPLICIT_AFFINITY capability**
implicit affinity use of, 45
**$CPBDEF macro**
capabilities symbolic values, 20
**CPU affinity**. *See also* CPU capabilities; scheduling; SMP systems.
characteristics and use, 43–45, 254
explicit. *See also* CPU$L_HARDAFF field.
acquired through $DIAGNOSE, 44
acquired through $PROCESS_AFFINITY, 43
acquired through SCH$SET_AFFINITY, 43
characteristics and use, 43
examples of use, 43–44
released at process deletion, 428
released through SCH$RELEASE_CAPABILITY, 42
implicit
acquired through $SET_IMPLICIT_AFFINITY, 45
acquiring, 45
characteristics and use, 45
effect on scheduling, 45
enabling, 275
mechanism, 45
initialized by EXE$CREPRC, 275
mismatch handling, 59–60
not compatible with DECthreads, 216
required by kernel threads, 43
requirements
as significant scheduling characteristic, 9
transition to RWCAP resulting from, 34
reset at image rundown, 44
system services that affect, 254
term definition, 43
**CPU capabilities**. *See also* CPU affinity; CPU$L_CAPABILITIES field; scheduling; SMP systems.
changing, 42
characteristics and use, 41–43, 254
CPB$V_PRIMARY, characteristics and use, 41
CPB$V_QUORUM, characteristics and use, 41
CPB$V_RUN, characteristics and use, 41
described in per-CPU database, 42
initialized by EXE$CREPRC, 275
mismatch handling, 59–60
not compatible with DECthreads, 216
released at process deletion, 428
removing, 59
required by kernel threads, 42
requirements
as significant scheduling characteristic, 9
transition to RWCAP resulting from, 34
reset at image rundown, 44
summarized in SCH$AL_CPU_CAP array, 42
system services that affect, 255
transitions triggered by, 39
user-defined, 42
uses, 41
**CPU (central processing unit)**. *See* per-CPU database; SMP systems; VMScluster systems.
**CPU ID**
system mask use of, 11
**CPU mutex**
use by SCH$REMOVE_CPU_CAP, 59
**CPU time limit expiration**
checked at quantum end, 56, 181
**CPU$L_CAPABILITIES field**
current CPU capabilities, 59
definition and use, 20, 42
**CPU$L_CPUID_MASK field**
definition and use, 19
**CPU$L_CURKTB field**
address of kernel thread in CUR state, 32
definition and use, 19
**CPU$L_CURPCB field**
definition and use, 19

**CPU$L_CUR_PRI field**
definition and use, 19, 29
**CPU$L_HARDAFF field**
definition and use, 20, 43
image rundown effect on, 44
process deletion effect on, 44
**CPU$L_PHY_CPUID field**
definition and use, 19
**CPU$L_ZEROED_PAGE_SPTE field**
definition and use, 19
**CPU$L_ZEROED_PAGE_STATE field**
definition and use, 20
**CPU$L_ZEROED_PAGE_VA field**
definition and use, 20
**CPU$Q_ASN field**
definition and use, 25
**CPU$Q_ASNSEQ field**
definition and use, 25, 26
**CPU$Q_CAPABILITIES field**
definition and use, 20
**CPU$Q_IDLE_LOOP_COUNT field**
definition and use, 20
**CPU$Q_SYS_HWPCB field**
definition and use, 20
**CPU$Q_ZEROED_PAGE_COUNT field**
definition and use, 20
**CPUEXIT bugcheck**. *See* bugchecks.
**CPUSANITY bugcheck**. *See* bugchecks.
**CPUSPINWAIT bugcheck**. *See* bugchecks.
**$CPU_CAPABILITIES (Modify CPU User Capabilities system service)**
characteristics and use, 42
overview, 254
**crash**. *See* bugchecks.
**crash dumps**. *See* bugchecks; errors.
**Create Process system service**. *See* $CREPRC.
**Create Virtual Address Space system service**. *See* $CRETVA.
**$CREATE_KERNEL_THREAD system service**
control flow, 170–173
operations, 169
**$CREATE_REGION_64 system service**
use in initiating multithreading, operations, 164
**$CREPRC (Create Process system service)**. *See also* process creation.
arguments passed to, by
DCL, 463
job controller process, 440 (table)
NETSACP, 452
NETACP, 452
base priority initialization by, 28
control flow, 265–277
establishing quotas for new process, control flow, 277–278
operations, overview, 7, 263
requested by
job controller, 441
job controller process, 439
NET$ACP, 452
NETACP, 452
**$CRETVA (Create Virtual Address Space system service)**. *See* address space; virtual address space.
**$CRMPSC (Create and Map Section system service)**. *See* global sections; process sections.
**CSID (cluster system ID)**
EPID updated from, 283
field contents explained, 123
VMScluster node search use of, 231
**CTL$AL_CLICALBK cell**
address of CLI callback service routine, 453
**CTL$AQ_STACK array**
address of user stack high end, 339
initial values of per kernel thread stack pointers, 162
**CTL$AQ_STACKLIM array**
lowest limit of per kernel thread stacks, 162
**CTL$AR_IM_SEMAPHORE cell**
address of inner mode semaphore, 165, 186
**CTL$A_IMGREG cell**
address of P1 space image region for nonresident image sections, 359
**CTL$A_INITIAL_PKTA array**
address of initial kernel thread PKTA, 162
**CTL$GB_MSGMASK cell**
default message flags, 292
use by LOGINOUT, 447
**CTL$GL_BPD_PTR cell**
bound procedure descriptor virtual address space, 347
**CTL$GL_CTLBASVA cell**
initial low-address end of P1 space, 291, 293
use by LOGINOUT, 449
**CTL$GL_FINALSTS cell**
final image status, 366
**CTL$GL_FIXUPLNK cell**
fixup section list, 349
**CTL$GL_GETMSG cell**
address of message vector, 292
**CTL$GL_IMGHDRBF cell**
address of image header buffer, 340, 374
**CTL$GL_INITIAL_EPID cell**
EPID of process, 291
**CTL$GL_INITIAL_KTB cell**
address of initial kernel thread KTB, 156, 291
**CTL$GL_INITIAL_PID cell**
identifies initial kernel thread, 154
IPID of process, 291
**CTL$GL_KT_AREA_PAGES cell**
size of PKTA area, 165

**CTL$GL_KT_COUNT cell**
number of kernel threads, 156, 291
**CTL$GL_KT_HIGH cell**
number of highest KTB vector entry, 156
**CTL$GL_MULTITHREAD cell**
initialized by EXESINIT_MULTITHREAD, 165
nonzero for a multithreaded process, 244
number of possible KTBs, 156
**CTL$GL_PCB cell**
address of PCB, 289
**CTL$GL_PHD cell**
address of P1 window to PHD, 289
**CTL$GL_PSTFLAGS cell**
P1 process start flag, 294
**CTL$GL_SUBSYS_RIGHTS cell**
image rights list segment for protected subsystems, 376
**CTL$GL_USRUNDWN cell**
address of kernel mode image rundown routine tables, 292
kernel mode rundown routine list, 378
**CTL$GL_USRUNDWN_EXEC cell**
address of executive mode image rundown routine tables, 292
executive mode rundown routine list, 371
**CTL$GL_WSL cell**
address of P1 window to working set list, 289
**CTL$GQ_EXH_MUTEX cell**
synchronizing access to
exit control blocks, 365
exit handler listheads, 365
**CTL$GQ_EXIT_THREAD cell**
use by EXE$EXIT, 366
**CTL$GQ_ISTART cell**
use by EXE$KRNDWN, 372
**CTL$GQ_KT_REGION_BASE cell**
virtual address of kernel threads private area, 162, 165
**CTL$GQ_KT_REGION_ID cell**
kernel thread private area ID, 162, 165
**CTL$GQ_LOGIN cell**
login time, 292
**CTL$GQ_PKTAVEC cell**
addresses of PKTAs, 162
**CTL$GQ_PRCTERM_LISTHEAD cell**
context block queue, 373
initializing, 291
**CTL$GQ_PROCPRIV cell**
definition and use, 374
process privilege mask, use and routines that manipulate it, 380(table)
use by LOGINOUT, 447
**CTL$GQ_PSTART cell**
login time, 292
**CTL$GQ_TM_CALLBACKS cell**
nonzero if user mode thread manager has been registered, 165
**CTL$GT_CLINAME cell**
use by LOGINOUT, 449
**CTL$GT_CLI_NAMESTRING cell**
use by LOGINOUT, 449
**CTL$GT_SPAWN_CLI cell**
use by LOGINOUT, 449
**CTL$T_USERNAME cell**
use by LOGINOUT, 446
**CTRL/Y**
handling during image execution, 364
processing, image termination operations, 459
**CUR (current state).** *See also* CPU$L_CURKTB field; CPU$L_CURPCB field; kernel thread states; wait states.
characteristics and transitions, 32–34
kernel thread, pointer to, in per-CPU database, 15
placing into wait state, 46–52, 48(fig.)
preemption mechanism, 60–63
transitions
from CUR to COLPG, 37
from CUR to FPG, 36
from CUR to HIB, 35
from CUR to other states, 32
from CUR to PFW, 36
**current state.** *See* CUR.
**CWPS (clusterwide process services).** *See also* VMScluster systems.
creating processes on other nodes, 269
extending process control and information system services, 225
locating processes on VMScluster nodes, 228
remote request processing, 228
servicing requests for, 230(fig.)
use by
EXE$DELPRC, 419
EXE$FORCEX, 253
EXE$GETJPI, 240–241
EXE$NAM_TO_PCB, 228
EXE$RESUME, 252
EXE$SETPRI, 255
EXE$SUSPND, 247
EXE$WAKE, 245
**CWPS$GETJPI routine (CWPS_GETJPI module)**
dispatching $GETJPI requests to other VMScluster nodes, 240
**CWPS$GETJPI_PSCAN routine (CWPS_GETJPI module)**
dispatching $GETJPI requests to other VMScluster nodes, 240
**CWPS$SRCV_GETJPI_AST routine ([SYS-LOA]CWPS_SERVICE_RECV module)**
operations, 240

**$CWPSJPI macro**
$GETJPI requests defined by, 232, 240
**CWPSSRV (clusterwide process services data block)**
dequeued at process deletion, 429
structure linkage with PSCANCTX and PHD, 232 (fig.)
use by EXE$CREPRC, 265, 269
**$CWPSSRV macro**
passing information to a remote VMScluster node, 232, 240
**CWPSSRV$L_EXT_OFFSET field**
offset to a CWPSSRV extension, 232
**CWPS_GETJPI module.** *See* CWPS$GETJPI routine; CWPS$GETJPI_PSCAN routine.
**cycle counters.** *See* PCC.

**D floating-point operands**
VEST handling of, 386
**$DACEFC (Disassociate Common Event Flag Cluster system service)**
control flow, 89–90
**data alignment trap fixup register.** *See* DATFX register.
**data incoherency.** *See* synchronization.
**data parameter area (translated image)**
follows bag routine procedure descriptor, 405
layout, 395 (fig.)
**data sections**
compressed, term definition, 307
**data structures**
affected by class scheduling, 78
AST-related, for multithreaded processes, 196–197
built by VEST, 394
DECthreads, 213–215
figures that describe, xiv–xv
images, 311–325
kernel threads, relations among, 155 (fig.)
lock management, characteristics and field definitions, 109–125
related to multithreading, 153–163
related to $PROCESS_SCAN, 229–232
scheduling, characteristics and field definitions, 10–20
that describe known images, 319–325
that describe process context, 3 (fig.)
**database.** *See* per-CPU database; scheduler database.
**date.** *See* time.
**DATFX (data alignment trap fixup) register**
as part of hardware context, 21
**DBLERR bugcheck.** *See* bugchecks.
**DCL (Digital Command Language) CLI**
command processing loop
commands handled by internal procedures, 456 (table)
control flow, 455 (fig.)
operations, 454 (table)
commands
interactive subprocess creation and connection, 461
exit handler, control flow, 458–459
image initiation
argument list passed to image, 458
operations, 457–458
image processing and, 452–468
initialization, operations, 453–454
**DCL exit handler.** *See* DCL$EXITHAND.
**DCL symbols**
spawned subprocess inherited from creator, 464
**DCL$ATTACH routine ([DCL]SPAWN module)**
control flow, 465–466
**DCL$EXITHAND (DCL exit handler)**
control flow, 458–459
supervisor mode exit handler, 453
**DCL$EXTIMAGE routine ([DCL]IMAGEXECT module)**
image initiation, control flow, 457–458
**DCL$SPAWN routine ([DCL]SPAWN module)**
control flow, 462–465
**DCL$STARTUP routine ([DCL]INITIAL module)**
operations, 453–454
**$DCLEXH (Declare Exit Handler system service)**
operations, 364
**[DCL]IMAGEXECT module.** *See* DCL$EXITHAND; DCL$EXTIMAGE routine.
**[DCL]INITIAL module.** *See* DCL$STARTUP routine.
**[DCL]SPAWN module.** *See* DCL$ATTACH routine; DCL$SPAWN routine.
**DEADLOCK module.** *See* LCK$SEARCHDLCK routine; LCK$SRCH_RESDLCK routine.
**deadlocks.** *See also* locks; resources; synchronization.
avoiding with pixscan boost, 66
conversion
detecting, method for, 143
preventing locks from participating in search, 142
victim selection, 148–149
detecting, 143–148
system-owned lock queue restrictions resulting from, 136
$ENQ argument restricting lock participation in search, dangers of, 142
handling, 141–149
phases, 141
multiple resource
detecting, 143–146, 144 (fig.)

preventing locks from participating in search, 142
recursive algorithm for verifying, 145 (fig.)
search for, example, 148 (fig.)
victim selection, 148–149
reasons for, 141
search for
criteria for initiating, 141
triggering for lock requests, lock timeout queue role in, 142
term definition, 109
unsuspected
criteria for becoming a suspected deadlock, 146
detecting, 146–147
preventing accidental detection of, 147 (fig.)
term definition, 146
**DEADLOCK_WAIT parameter (SYSGEN)**
deadlock search initiation controlled by, 141
**DEBUG command**
operations, 460
**debugger**
bootstrap, name for $IMGSTA system service, 362
calling during image execution, 363
mapped by $IMGSTA, 362
**debugging**. *See* bugchecks; errors.
**DECdtm**
stream associated with, SYS$RMSRUNDWN operations, 369
**Declare Exit Handler system service**. *See* $DCLEXH.
**DECmigrate**
characteristics and use, 383
**DECnet**. *See* VMScluster systems; *Consult* DECnet for OpenVMS Guide to Networking; DECnet for OpenVMS Networking Manual.
**DECthreads**. *See also* user mode thread manager.
accepting unblocking upcalls, 217
data structures, 213–215
image exit handling, 220
implemented as shareable image, 212
initializing, 212–213
null thread
awakening the kernel thread running, 213
characteristics and use, 212–213
overview, 211–212
PKTA user-writable fields used for communication with the executive, 162
role at event flag waits, 82
scheduling, 216–218
user mode threads, 2, 178
scheduler database, 216
synchronization, 216
user mode threads
managing, 2, 151
multiplexing, 216
scheduling, example, 218, 219 (fig.)
unblocking, 211
user stack created by, 215
**DEFPRI parameter (SYSGEN)**
job controller's $CREPRC default, 268
normal kernel thread priority defined from, 31
**Delete Common Event Flag Cluster system service**. *See* $DLCEFC.
**DELETE kernel mode AST procedure**
control flow, 421–426
single-image process, control flow, 438
**Delete Process system service**. *See* $DELPRC.
**DELETE_IN_SYS_CONTEXT routine (SYSDELPRC module)**
control flow, 428–430
**$DELETE_KERNEL_THREAD system service**
kernel thread deletion, control flow, 174–175
routines that request, 173–174
**$DELETE_REGION_64 system service**
terminating multithreading, operations, 168
**$DELPRC (Delete Process system service)**. *See also* process deletion.
control flow, 417–421
operations, 417
**demand zero sections**
EISD that describes, 332–334
initialization of EISD and PTEs for, 334 (fig.)
**$DEQ (Dequeue Lock Request system service)**
control flow, 137–139
**Dequeue Lock Request system service**. *See* $DEQ.
**$DEREGISTER_THREAD_MANAGER system service**
control flow, 168–169
**DETACH (create detached processes privilege)**. *See also* privileges.
use by EXE$CREPRC, 268, 277, 279
**device affinity**. *See* CPU affinity.
**Device lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 474–476
disposing of, 475
**device locks**. *See* spinlocks.
**DIAGNOSE (diagnose devices privilege)**. *See also* privileges.
acquiring or removing explicit affinity, 44
**$DIAGNOSE (Diagnose system service)**
explicit affinity, 44
**Diagnose system service**. *See* $DIAGNOSE.
**directory**
default, copied at process creation, 273
**directory (lock management)**
distributed, VMScluster root resources, characteristics and use, 124
**directory node (lock management)**
characteristics and use, 123–124

**directory node (lock management)** *(continued)*
operations in response to lock request for nonlocal resource, 124
resource tree, 124
**dirty zero**
term definition, 387
**DISABLE_UPCALLS parameter (SYSGEN)**
use by EXE$REGISTER_THREAD_MANAGER, 166
**Disassociate Common Event Flag system service.** *See* $DACEFC.
**Disk Quota Cache lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 484
**disks.** *See* bad blocks.
**$DISMOU (Dismount Volume system service)**
lock use by, characteristics and use of each lock, 477–478
**Dismount lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 477–478
**Dismount Volume system service.** *See* $DISMOU.
**$DLCEFC (Delete Common Event Flag Cluster system service)**
control flow, 90
**DLCKEXTRASTK parameter (SYSGEN)**
limiting resource tree maximum depth with, 145
**dormant processes.** *See* kernel threads—dormant.
**DORMANTWAIT parameter (SYSGEN)**
pixscan mechanism use of, 67
**driver dispatch table.** *See* DDT.
**driver prologue table.** *See* DPT.
**Driver-Loading and Autoconfiguration lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 507
**drivers.** *See* class drivers; device drivers; port drivers; terminal drivers.
**[DRIVER] facility**
device driver source modules contained in, xii
**dump.** *See* crash dumps.
**dynamic remastering**
resource trees, 124

**EFN$C_ENF (event flag 128)**
EXE$READEF actions, 102
preventing concurrent use of event flags, 102–103
SCH$POSTEF actions, 101
$SYNCH handling of, 84
system service use of, 83–84
**EIHD (OpenVMS Alpha image header).** *See also* image header.
field definitions, 313–314
layout, 312 (fig.)
**$EIHDDEF macro**
image header fixed part definition, 312
**EIHI (image ident area)**
characteristics and use, 314–315
field definitions, 314–315
layout, 315 (fig.)
**EISD (image section descriptor)**
characteristics and use, 300, 315–317
demand zero, 315
field definitions, 315–317
global, 316, 334
identifies section shareable image, 317
initialization for
demand zero section, 334 (fig.)
global section, 333 (fig.)
process-private section, 331 (fig.)
layout, 316 (fig.)
processing
by EXE$IMGACT, 330–334
while activating shareable image, 337–338
section, 316
that describes
demand zero section, 332–334
fixup section, 332
global section, 330–332
process-private section, 330
types, 315–317
**EISD$V_VECTOR bit (EISD$L_FLAGS field)**
identifies private or global section, 337
**emulation.** *See* instruction emulation.
**$END_MULTITHREAD system service**
control flow, 167–168
requested by DELETE, 421
**$ENQ (Enqueue Lock Request system service)**
control flow
when parent resource does not exist, 128–129
when parent resource exists, 127–128
creating RSB for new resource, 130
$ENQW
asynchronous form of $ENQ, 131
establishing default process resource domain, 128
granting locks, operations, 133–135
kernel thread state change actions, 35, 36
lock conversion, operations, 132–133
operations, 127–131
**Enqueue Lock Request system service.** *See* $ENQ.
**entry points**
names, xiii
**environment variables.** *See* $GETENV.
**EOF lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 488–489
**EPID (extended process identifier)**
characteristics and use, 225, 279–280, 282–283
construction, 282

copying during subprocess creation, 271
IPID use compared to, 225
layout, 282 (fig.)
LKB field that contains, 119
locating target kernel thread with, 227
negative, as wildcard indicator, 225
not constant for a user mode thread, 215
obtaining next, in $GETJPI wildcard, 242–243
routines that transform and manipulate, 279–280
transforming IPID into, 154
unique for kernel thread on a VMScluster, 154
updated from CSID, 283
use in subprocess SPAWN and ATTACH, 462
**equivalence names**. *See* logical names.
**ERNDWN (Image Rundown system service)**. *See also* image rundown.
control flow, 371
rundown routines invoked by EXE$RUNDWN, 371
**errors**. *See also* bugchecks; deadlocks; error log; exceptions; interrupts; IPL; traps.
conditions, signaled during image execution, 364
**event flag wait services**
kernel mode, operations, 92–94
mode of caller, operations, 91–92
using unique event flags, 103
**event flag wait state**. *See also* CEF; LEFO.
context for, 53
KTB fields that identify flags waited for, 14
KTB fields related to, 87
placing kernel threads into, 90
removing kernel threads from
CEF, 89, 101
LEF or LEFO, 101
**event flags**. *See also* CEF; common event flags; EFN$C_ENF; event flag wait state; LEFO; local event flags; synchronization; wait states.
AST queuing constraints on use of, 103
avoiding ambiguous use of, 82
characteristics and use, 81–82
clearing, 102
interlocked operation, 104
practices to avoid, 103
clusters 0 and 1, location, 83
clusters 2 and 3, location, 85
concurrent uses of the same, 82
correct use of, 102–104
default flag, 83
ending kernel thread wait, 101
flag 0, 83
flag 128. *See* EFN$C_ENF.
indefinite wait, example, 103–104
kernel threads waiting for, 90–98
KTB fields related to, 86–87, 87 (fig.)
meaning of, 81
PCB fields related to, 86–87, 87 (fig.)
preventing concurrent use of, 102–103
reading, 102
reserved for system use, 83
reserved through LIB$GET_EF, 102
setting, 100–102, 241
event causing kernel thread state change, 60
by EXE$GETJPI, 239
by EXE$GETLKI, 140
in a multithreaded process, 210
reported when dissociating from a common event flag cluster, 89
when a lock is dequeued, 139
when a lock is granted, 130, 134
system services that use, 82
use as process state, 91
waiting for, 91–98
in a multithreaded process, 210
**events**
with no associated event flag, 83–84
**exception handlers**. *See* condition handlers.
**EXCEPTION module**. *See* MMG$SWAP_PTBR routine.
**exceptions**. *See also* condition handlers; conditions.
Alpha precision, compared to VAX, 387
fault-on-execute, meaning and use, 413
imprecise
VEST handling of, 387
**EXE$ASCEFC routine (SYSASCEFC module)**
creating common event flag clusters, control flow, 88–89
**EXE$CALLBACK_JACKET routine (TM_SUPPORT module)**
control flow, 208–209
**EXE$CATCH_ALL routine (PROCSTRT module)**
established as last chance handler, 296
established as oldest frame-based handler, 296
**EXE$CHECKS_PROCESS_ACCESS routine (SYSPCNTRL module)**
operations, 227
**EXE$CHECK_UPCALL routine (TM_SUPPORT module)**
checking if upcall possible, 190, 194
**EXE$CHK_WAIT_BHVR routine (RSE module)**
proactive memory reclamation, control flow, 55
**EXE$CLREF routine (SYSEVTSRV module)**
clearing event flags, operations, 102
**EXE$CMODEXEC routine (SYSTEM_SERVICE_DISPATCHER module)**
acquiring inner mode semaphore, 188
releasing inner mode semaphore, 193
**EXE$CMODKRNL routine (SYSTEM_SERVICE_DISPATCHER module)**
acquiring inner mode semaphore, 188
releasing inner mode semaphore, 193

**EXE$CREATE_KERNEL_THREAD routine (TM_SUPPORT module)**
control flow, 170–173
operations, 169
**EXE$CREPRC routine (SYSCREPRC module)**
control flow, 265–277
establishing quotas for new process, control flow, 277–278
operations, overview, 263
**EXE$CVT_EPID_TO_IPID routine (SYSPCNTRL module)**
purpose, 279
**EXE$CVT_EPID_TO_KTB routine (SYSPCNTRL module)**
purpose, 279
**EXE$CVT_EPID_TO_PCB routine (SYSPCNTRL module)**
purpose, 280
**EXE$CVT_IPID_TO_EPID routine (SYSPCNTRL module)**
purpose, 280
**EXE$CVT_IPID_TO_KTB routine (SYSPCNTRL module)**
purpose, 280
**EXE$CVT_IPID_TO_PCB routine (SYSPCNTRL module)**
purpose, 280
**EXE$DACEFC routine (SYSASCEFC module)**
dissociating from an event flag cluster, control flow, 89–90
**EXE$DEBIT_BYTCNT routine (EXSUBROUT module)**
use in byte count quota wait, 54
**EXE$DELETE_KERNEL_THREAD routine (TM_SUPPORT module)**
kernel thread deletion, control flow, 174–175
**EXE$DELPRC routine (SYSDELPRC module)**
control flow, 417–421
**EXE$DEQ routine (SYSENQDEQ module)**
control flow, 137–139
**EXE$DEREGISTER_THREAD_MANAGER routine (TM_SUPPORT module)**
control flow, 168–169
**EXE$DLCEFC routine (SYSASCEFC module)**
deleting permanent event flag clusters, control flow, 90
**EXE$END_MULTITHREAD routine (TM_SUPPORT module)**
control flow, 167–168
**EXE$ENQ routine (SYSENQDEQ module)**
control flow
when parent resource does not exist, 128–129
when parent resource exists, 127–128
creating RSB for new resource, 130
establishing default process resource domain, 128
granting locks, operations, 133–135
lock conversion, operations, 132–133
operations, 127–131
**EXE$ERNDWN routine (SYSRUNDWN module)**
control flow, 371
**EXE$EXIT routine (SYSEXIT module)**
control flow, 366–368
exit handler list processing, control flow, 368–369
multithreaded process, operations, 436
upcalls, operations, 205
**EXE$EXIT_INT routine (SYSEXIT module)**
control flow, 366–368
multiple-image process, control flow, 468
single-image process, control flow, 436–438
**EXE$FANDLE_RUNDOWN routine (SYSFASTIO module)**
operations, 372
**EXE$FORCEX routine (SYSFORCEX module)**
controlling processes with, 252–253
upcalls, operations, 207
**EXE$FORCE_QUANTUM_END routine (TM_SUPPORT module)**
user mode thread quantum end, operations, 210
**EXE$GETJPI routine (SYSGETJPI module)**
control flow, 235–240
**EXE$GETLKI routine (SYSGETLKI module)**
operations, 139–140
**EXE$GL_ABSTIM cell**
number of seconds since bootstrap, 142
use in lock timeout processing, 142
**EXE$GL_ABSTIM_TICS cell**
copied to KTB$L_WAITIME, 49
deducted from KTB$L_QUANT, 56
**EXE$GL_CEBMTX cell**
synchronizing access to CEB list, 85
**EXE$GL_RSDM_MUTEX cell**
RSDM mutex for resource domain data structure access, 140
**EXE$GL_SYSID_LOCK cell**
KFE locks use of, 496
**EXE$GL_USRUNDWN_EXEC cell**
definition and use, 371
**EXE$GQ_RSDM_LIST cell**
listhead for RDDBs, 115
**EXE$HIBER routine (SYSPCNTRL module)**
control flow, 244
upcalls, operations, 206
**EXE$HIBER_INT routine (SYSPCNTRL module)**
control flow, 244–245
proactive memory reclamation, operations, 54
**EXE$IMGACT routine (SYSIMGACT module).** *See also* image activation.
activating
shareable image, control flow, 335–338
shareable image, example, 343–344

simple main image, control flow, 326–335
end processing, control flow, 338–340
known as image activator, 326
procedures called by, 326
**EXE$IMGFIX routine (SYSIMGFIX module)**
address fixup
code, 355
linkage pair, 356
linkage pair with procedure signature blocks, 356–357
longword and quadword, 354
calling shareable images initialization routines, 360
operations, 359–360
testing for privileged shareable images activation, 360
**EXE$IMGSTA routine (SYSIMGSTA module)**
multi-image process, control flow, 466–468
operations, 362–363
single-image process, 467 (fig.)
control flow, 435–436
**EXE$INFORM_TM_AST routine (TM_SUPPORT module)**
control flow, 208
executive routines that call, 206–207
**EXE$INFORM_TM_CALL routine (TM_SUPPORT module)**
control flow, 207–208
executive routines that call, 205–206
**EXE$INIT_MULTITHREAD routine (TM_SUPPORT module)**
control flow, 164–165
**EXE$KERNEL_WAIT_PS routine (MUTEX module)**
use in job resource wait, 54
**EXE$KRNDWN routine (SYSRUNDWN module)**
control flow, 371–377
single-image process, operations, 438
**EXE$KTHREAD_DELETE_CALLBACK routine (TM_SUPPORT module)**
kernel thread deletion, control flow, 177–178
**EXE$KTHREAD_DELETE_SELF routine (TM_SUPPORT module)**
kernel thread deletion, control flow, 175–177
**EXE$KTHREAD_START routine (TM_SUPPORT module)**
operations, 173
**EXE$NAM_TO_PCB routine (SYSPCNTRL module)**
arguments, 226–227
control flow, 227–228
PID and privilege checking with, 224
process deletion, control flow, 417–419
**EXE$NATIVE_TO_TRANSLATED routine (SYSJACKET module)**
control flow, 402–403
**EXE$POSTEF_AST routine (POSTEF module)**
operations, 211
**EXE$PRCDELMSG routine (ACCOUNT module)**
operations, 425
**EXE$PROCESS_SCAN routine (PROCESS_SCAN module)**
control flow, 233–234
data structures used by, 229–232, 231 (fig.)
**EXE$PROCSTRT routine (PROCSTRT module)**
control flow, 290–296
environment, 290
image initiation, argument list passed to image, 458
multi-image process, 467 (fig.)
operations, overview, 263–264
single-image process, control flow, 434–435
**EXE$PSCAN_LOCKCTX routine (PROCESS_SCAN module)**
operations, 235
**EXE$PSCAN_NEXT_PID routine (PROCESS_SCAN module)**
operations, 236–237
**EXE$READEF routine (SYSEVTSRV module)**
reading event flags, operations, 102
**EXE$REGISTER_THREAD_MANAGER routine (TM_SUPPORT module)**
control flow, 165–167
**EXE$REI_INIT_STACK routine (REI module)**
use at process creation, 295
use by LOGINOUT, 450
**EXE$RESCHED routine (SYSPARPRC module)**
control flow, 256–257
**EXE$RESUME routine (SYSPCNTRL module)**
operations, 252
**EXE$RUNDWN routine (SYSRUNDWN module)**
operations, 370
**EXE$SCHDWK routine (SYSSCHEVT module)**
operations, 245
**EXE$SETAST routine (SYSASTCON module)**
upcalls, operations, 205
**EXE$SETPRI routine (SYSSETPRI module)**
operations, 255–256
**EXE$SETPRN routine (SYSPCNTRL module)**
operations, 257
**EXE$SETPRV routine (SYSSETPRV module)**
operations, 381
**EXE$SINGLE_THREAD routine (TM_SUPPORT module)**
arguments, 202
control flow, 202–203
**EXE$SINGLE_THREAD_INT routine (TM_SUPPORT module)**
operations, 203
**EXE$SOFT_SUSPND_INT routine (SYSPCNTRL module)**
operations, 251–252

**EXESSS_RELEASE_THREAD routine (SYSTEM_SERVICE_IMSEM module)**
control flow, 194–195
**EXESSS_WAIT_THREAD routine (SYSTEM_SERVICE_IMSEM module)**
control flow, 192
**EXE$SUSPND routine (SYSPCNTRL module)**
AST queuing during process suspension, 247
control flow, 247–249
**EXE$SYNCH routine (SYSSYNCH module)**
mode of caller $SYNCH procedure, 94
**EXE$SYNCH_INT routine (SYSSYNCH module)**
control flow, 97
**EXE$SYNCH_LOOP routine (SYSSYNCH module)**
synchronous system service return path, control flow, 94–96
upcalls, operations, 206
**EXE$TIMEOUT routine (TIMESCHDL module)**
deadlock search initiated by, 142
pixscan mechanism called by, 66
**EXE$TRANSLATED_TO_NATIVE routine (SYSJACKET module)**
control flow, 403–404
**EXE$WAIT routine (SYSWAIT module)**
event flag wait, common code, control flow, 92–94
proactive memory reclamation, operations, 54
**EXE$WAITFR routine (SYSWAIT module)**
upcalls, operations, 206
**EXE$WAKE routine (SYSPCNTRL module)**
operations, 245–246
**EXE$WFLAND routine (SYSWAIT module)**
upcalls, operations, 206
**EXE$WFLAND_COMMON routine (SYSWAIT module)**
event flag wait, common code, control flow, 92
**EXE$WFLAND_INT routine (SYSWAIT module)**
event flag wait, common code, control flow, 92
**EXE$WFLOR routine (SYSWAIT module)**
upcalls, operations, 206
**EXE$WFLOR_COMMON routine (SYSWAIT module)**
event flag wait, common code, control flow, 92
**EXE$WFLOR_INT routine (SYSWAIT module)**
event flag wait, common code, control flow, 92
**executable images**
building, 302–303
installed /RESIDENT, activating, 307
installed /SHARED, virtual address space, 306
term definition, 301
typically require no address relocation, 349
**execution**
kernel thread priorities, concepts and mechanisms, 27
**execution contexts**
characteristics and use, 2, 151
provided by kernel threads, 223
**executive**
components, xii
executing images, 299
jacketing
initiated by, 401
prohibited by, 401
lock use by, characteristics and use of each lock, 473–476
term definition, xii
upcall use by, 204
**executive images.**
special kind of shareable image, 302
term definition, xii
**executive mode**
logical names, created by LOGINOUT, 449
stack pointer, as part of hardware context, 20
**executive mode stack.** *See* executive stack.
**executive routines**
entry points, xiii
**executive stack**
pages, protection, 161
**EXEC_RUNDOWN_AST routine (SYSDELPRC module)**
operations, 421
**$EXIT (Exit system service).** *See also* image rundown.
control flow, 366–368
exiting main image of process, 436
multithreaded process, operations, 436
operations, 364
requested following
image return to EXE$PROCSTRT, 296
rundown routines invoked by EXE$RUNDWN, 367
terminating processes with, 252
**EXIT command**
operations, 460
**exit control block**
characteristics and use, 364–365
layout, 365 (fig.)
**exit handlers.** *See also* RMS exit handler.
compared to rundown routines, 377
executive mode, declared by EXE$PROCSTRT, 295
images, characteristics and operations, 364–365
list processing, 368, 369 (ex.)
listhead array, layout, 365 (fig.)
RMS, declared by EXE$PROCSTRT, 295
supervisor mode
canceling by LOGINOUT, 469
declared by CLI, 453, 457
user mode, not entered after STOP command, 461
**Exit system service.** *See* $EXIT.
**EXIT_CALLBACK routine (SYSEXIT module)**
operations, 367

single-image process, control flow, 436–438, 468
**$EXIT_INT system service**
CLI exit handler called by, 458
**Expand Region system service**. *See* $EXPREG.
**explicit affinity**. *See* CPU affinity—explicit.
**EXSUBROUT module**. *See* EXE$DEBIT_BYTCNT routine.
**extended process ID**. *See* EPID.
**Extended QIO Processor**. *See* XQP.
**Extent Cache lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 483–484
**EXTRACPU parameter (SYSGEN)**
quantum expiration use of, 56

**F11BXQP.EXE**. *See* XQP.
**facilities**
codes, 472 (table)
source modules contained in, xii
**fandle**
term definition, 372
**Fast-IO handle**. *See* fandle.
**FATALEXCPT bugcheck**. *See* bugchecks.
**faults**. *See* debugging; errors; page faults; traps.
**figures**
conventions, xiv–xv
that describe data structures, xiv–xv
**FILCNTNONZ bugcheck**. *See also* bugchecks.
generated by DELETE, 423
**File Access Arbitration lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 480–481
**File ID Cache lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 483
**File lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 486–488
**file protection**
default, copied at process creation, 272
**File Serialization lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 481–482
**file system**. *See also* logical names; RMS; XQP.
lock use by, characteristics and use of each lock, 478–485
**files**
closing during process deletion, 422
open, KFE field that specifies WCB for, 320
**Files-11 Extended QIO Processor**. *See* XQP.
**fixed-length lists**. *See* IMCB; KRP; PQB.
**fixup section**
address relocation information contained in, 310
characteristics and use, 300, 349, 350–351
EIHD field that contains address of, 313
EISD that describes, 332
IMCB field that contains address of, 319
layout, 350 (fig.)
list, processed by EXE$IMGFIX, 360
not shareable, 309
relocation records
characteristics and use, 352–353
layout, 353 (fig.)
subset
address relocations and fixups described by, 349
characteristics and use, 324
created by Install utility, 346
KFERES field that contains address of, 324
**fixups**. *See also* address fixup.
characteristics and use, 399
page protection, 357–358, 358 (fig.)
**FKB (fork block)**. *See* fork processes; kernel processes.
**floating-point register and execution data structure**. *See* FRED.
**floating-point registers**
as part of hardware context, 3, 20
saved in PHD, 3
**Force Exit system service**. *See* $FORCEX.
**$FORCEX (Force Exit system service)**
controlling processes with, 252–253
**$FORCE_QUANTUM_END system service**
user mode thread quantum end, operations, 210
**fork block**. *See* FKB.
**fork processes**. *See also* IPL; synchronization.
lock management system, interface to, 137
**forking**. *See* fork processes.
**Forms Cache lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 505
**FPG (free page wait state)**. *See also* kernel thread states; wait states.
context for, 53
transitions
from FPG to COM or COMO, 37
to FPG from CUR, 36
**frame-based condition handlers**. *See* condition handlers.
**FRED (floating point register and execution data structure)**
layout in PHD, 160 (fig.)
scheduling-related fields, 18 (fig.)
**FRED (floating-point register and execution data structure)**
characteristics and field definitions, 156–161
characteristics and use, 4
double-mapped, 161
hardware context saved in, 21
HWPCB contained in, 11
scheduling-related fields, 17–18
substructure of PHD, 11, 156

**FRED (floating-point register and execution data structure)** *(continued)*
virtual address, 14
**FRED$L_FLAGS field**
definition and use, 161
scheduling-related FRED field, 18
**FRED$Q_ASN field**
copied from PCB$Q_PRVASN, 73
definition and use, 25
**FRED$Q_ASNSEQ field**
copied from
CPU$Q_ASNSEQ, 74
PCB$Q_ASNSEQ, 73
definition and use, 25, 161
**FRED$Q_ASTSR_ASTEN field**
initialized by EXE$CREATE_KERNEL_THREAD, 172
rescheduling interrupt service routine use of, 74
**FRED$Q_FPCR field**
floating-point control processor register contents, 18
**FRED$V_AST_PENDING bit (FRED$L_FLAGS field)**
definition and use, 18
use by SCH$INTERRUPT, 74
**FRED$V_SW_FEN bit (FRED$L_FLAGS field)**
definition and use, 18
**free page list**. *See* FPG.
**free page wait state**. *See* FPG.

**Get Environment Variable system service**. *See* $GETENV.
**Get Job/Process Information system service**. *See* $GETJPI.
**Get Lock Information system service**. *See* $GETLKI.
**$GETENV (Get Environment Variable system service)**
capabilities, 43
**$GETJPI (Get Job/Process Information system service)**
arguments, 234–235
control flow, 235–240
JPI$V_IGNORE_TARGET_STATUS, flag definition and use, 239
JPI$V_NO_TARGET_AST, flag definition and use, 238
JPI$V_NO_TARGET_INSWAP, flag definition and use, 238
locating next process or kernel thread, 242
remote nodes support, 240–241
search context filtering information, 229
special kernel mode ASTs, operations, 241–242
traditional wildcard support in, 242–243
used in conjunction with $PROCESS_SCAN, 229
wildcard requests
buffer use, 232
locating next process or kernel thread, 242–243
VMScluster node handling, 232
**$GETLKI (Get Lock Information system service)**
operations, 139–140
**Global Buffer Backing lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 493–494
**global buffer locks**
characteristics and use of each lock, 491–494
**Global Buffer Master lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 492
**Global Buffer Section lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 492–493
**global page table entry**. *See* GPTE.
**global page-file sections**. *See* global sections.
**global pages**. *See* global sections.
**global section table entry**. *See* GSTE.
**global sections**. *See also* memory management; sections.
creating
for images installed resident, 310
by Install utility, 330
EISD that describes, 330–332
initialization of EISD and PTEs for, 333 (fig.)
of installed images, 306
as interprocess communication mechanism, 261
name
determining, 304, 316, 328
EISD field that contains, 316
of shareable images, IMCB field that contains old-format name of, 318
use during image mapping, 309
**GPTE (global page table entry)**. *See* global sections.
**grain**
EIHD field that specifies size of, 314
term definition, 303
translated image, 392
**grant mode**. *See also* group grant mode; locks.
characteristics and use, 131
compared with conversion grant mode, 132–133
**granted queue**
removing LKBs from, 138
resource, listhead location in RSB, 112
**granularity hint region**
activating image resident in, 346
loading shareable image sections into, 305, 307
**GROUP (affect other processes in same group privilege)**. *See also* privileges.
required by EXE$NAM_TO_PCB, 227
required for
implicit affinity modification, 254
process affinity modification, 254

process capabilities modification, 255
**group grant mode**. *See also* grant mode; locks.
characteristics and use, 131
conversion request use of, 131
distinguished from conversion grant mode, 132
**GRPNAM (insert in group logical name table privilege)**. *See* privileges.
**GRPPRV (access group objects via system protection privilege)**. *See* privileges.
**GSMATCH linker option**
specifying number of global section major and minor IDs, 314
**GSTE (global section table entry)**. *See* global sections.

**H floating-point arithmetic**
not supported by Alpha, 387
**HALT bugcheck**. *See* bugchecks.
**halt codes**. *See* halts—reason for.
**halts**. *See* bugchecks; errors; machine checks.
restarts. *See* processors—restarting.
**hardware context**. *See also* execution contexts; HWPCB; SWPCTX PALcode routine.
characteristics and use, 20–24
distinguishing privileged from nonprivileged, 21
nonprivileged
restoring when placing a kernel thread back into execution, 24
saved on canonical kernel stack, 23
overview, 3–4
physical address of
KTB field that contains, 14
privileged, defined as contents of HWPCB, 21
routines that swap, 50–51
saving, 3
SWPCTX PALcode routine, control flow, 75–76
while placing a current kernel thread into wait state, 46
switching
characteristics and operations, 68
at kernel thread deletion, 76
by SCH$INTERRUPT, 74
**hardware privileged context block**. *See* HWPCB.
**HARD_SUSPND_AST routine (SYSPCNTRL module)**
kernel mode AST procedure, operations, 250–251
**/HEADER_RESIDENT qualifier**
known image installation, 304
**HIB (hibernate wait state)**. *See also* hibernation; HIBO; kernel thread states; wait states.
context for, 52
ended by $WAKE and $SCHDWK, 246
transitions
from HIB to COM, 35
to HIB from CUR, 35
**$HIBER (Hibernate system service)**
control flow, 244
kernel thread state change actions, 32, 35
**hibernate outswapped state**. *See* HIBO.
**hibernate outswapped wait state**. *See* HIBO.
**Hibernate system service**. *See* $HIBER.
**hibernate wait state**. *See* HIB.
**hibernation**. *See also* scheduling; suspension.
initial kernel thread, during process deletion, 421
placing processes in, 244–245
at process creation, 296
removing processes from, 245–246
suspension compared to, 247
**$HIBER_INT system service**
control flow, 244–245
**HIBO (hibernate outswapped wait state)**. *See also* HIB; hibernation; kernel thread states; wait states.
transitions
to HIBO from HIB, 35
from HIBO to COMO, 35
**hole**. *See* null page.
**huge pages**. *See* granularity hint region.
**HWPCB (hardware privileged context block)**
accessing, 22 (fig.)
during context switches, 21
contained in FRED, 11
hardware context contained in, 3
hardware context saved in, 21
layout, 22 (fig.)
per-CPU database contains, 20
PHD component, 21, 284
physical address, 14
scheduler database component, 11
swapper, adopting page tables, 286
updated by PALcode routine, 75–76

**I/O**. *See* resources; wait states.
**I/O channels**
images, IMCB field that specifies, 318
**I/O completion**
required for process deletion, 423
**I/O request packet**. *See* IRP.
**I/O status block**. *See* IOSB.
**I/O system services**. *Consult* OpenVMS System Services Reference Manual.
**IAC$GL_ICBFL cell**
IMCB lookaside list location, 317
**IAC$GL_IMAGE_LIST cell**
IMCB done list location, 317
**IAC$GL_WORK_LIST cell**
IMCB work list location, 317
**IDENTIFICATION linker option**
specifying image identification, 315

**idle loop**
executed by SCH$INTERRUPT, 76
per-CPU database fields used by, 19–20
responsibilities, 76
**IFB$L_PAR_LOCK_ID field**
definition and use, 487
**IJOBLIM parameter (SYSGEN)**
SYS$GL_IJOBCNT compared to, during process creation, 276
**image activation**. *See also* $IMGACT; $IMGFIX; $IMGSTA.
address fixups performed during, summary, 310
address relocation
$IMGFIX operations, 353
used during, 310
CLI, 329, 345
end processing, 338–340
executable images installed /RESIDENT, 307
image resident in granularity hint region, 346
images installed with shareable address data, 346
interoperable image, 347
known image, 325, 345
merged, 344–345
with message sections, 345
P0-only image, 345
privileged shareable image, 346–347
requested by
DCL, 457
EXE$PROCSTRT, 295
LOGINOUT, 448
resolving address references during, 358
from sequential devices, 345
shareable images, 335–338
installed /RESIDENT, 307
installed /SHARE=ADDRESS_DATA/RESIDENT, 307
simple main image, 326–335
special cases, 344–348
term definition, 299
types of, 325
uninstalled images, 303
**image activator**
address fixup performed by, 349
address relocation performed by, 349
buffers, local context block specifies, 319
EXE$IMGACT and procedures called are known as, 326
global section names determined by, 328
operations
message section, 337
privileged library vector, 337
system service intercept vector, 337
transfer address array, 339
processing IMCBs, 325
**Image Activator system service**. *See* $IMGACT.
**image body**
characteristics and use, 301
components, 301
**image control block**. *See* IMCB.
**image count**
tested prior to
$GETJPI completion, 241
LIB$SPAWN completion, 465
image counter
incremented at image rundown, 372
**image exit**. *See* $EXIT; image rundown.
**image header**. *See also* EIHD.
characteristics and use, 312–317
EISD
characteristics and use, 315–317
layout, 316 (fig.)
fixed part, characteristics and use, 313–314
fixup section, characteristics and use, 350–351
image ident area
characteristics and use, 314–315
layout, 315 (fig.)
IMCB field that specifies, 319
KFE field that specifies address of resident, 320
layout, 312 (fig.)
macros that define, 312 (fig.)
resident, advantages of, 304
transfer address array
characteristics and use, 360–362
layout, 361 (fig.)
**image ident area**. *See* EIHI.
**image reset**. *See* MMG$IMGRESET routine.
**image rights list segment**
characteristics and use, 376
**image rundown**. *See also* $ERNDWN; $EXIT; image activation; image activator; $KRNDWN; $RUNDWN.
common event flag clusters, processes automatically dissociated from, 90
distinguished from process rundown, 370
operations, 370–377
requested during process deletion, 423
term definition, 299
**Image Rundown system service**. *See* $ERNDWN; image rundown; $KRNDWN; $RUNDWN.
**image sections**
address fixup targets, 310
characteristics and use, 300
containing
NOWRT PSECTs, 350
relocated or fixed up addresses, 350
copy-on-reference, 310
creating, 300, 311
in image linked to 8 KB grain, 303
KFERES field that contains, 325
major and minor IDs, 315
overriding base address assignments, 358

placement, 324
protecting, 302
represented by relocation records, 353
shareable, loading into granularity hint region, 305, 307
system global sections created for, 306
types, 315
**Image Startup system service**. *See* $IMGSTA.
**images**. *See also* executable images; executive images; fixup section; image activation; $IMGACT; native images; privileged shareable images; shareable images; translated images.
activating, 302
address fixups, 354–357
arguments passed at initiation, 458
characteristics, 301–304
characteristics and use, 1
data structures, 311–325
EIHD field that identifies image type, 313
executable
term definition, 301
typically require no address relocation, 349
executing, 299
CLI commands that are handled without destroying, 456 (table)
exiting, 364–369
from executive mode at logout, 469
from user mode, 436
fixup section layout, 350 (fig.)
identification, 315
initializing shareable, computing proper order of, 340–343
initiation, 325
by CLI, 457–458
installed
/RESIDENT, activating, 307–308
/RESIDENT, fixups performed during image activation, 310
/RESIDENT, Install utility actions, 307
with shareable address data, address relocation and fixup processing
virtual address space, 306–308
installing, 302
example, 305
reasons for, 303–304
interoperable, 347
interrupted, state of, 460
interrupting with LIB$PAUSE, 460
known
characteristics and use, 304
data structures that describe, 319–325
linking, restrictions on, 359
main, term definition, 302
mapping
characteristics and use, 308–311
executable image with shareable images, 309 (fig.)
image installed resident with shareable address data, 310–311, 311 (fig.)
name
determining, 314
EIHI field that contains, 314
organization, 301 (fig.), 312 (fig.)
page size used by linker, 303
preactivating, 305, 358
privileged
rundown after CTRL/Y, 459
processes that execute single main, 434
protected, characteristics and use, 302
relocation records
characteristics and use, 352–353
layout, 353 (fig.)
rundown by CLI at initiation of new image, 457
simple main, image activator, control flow, 326–335
structure, 301
system service intercept, activating, 347
term definition, 1, 299
terminating
abnormally, 459–461
normally, 458–459
options, 458
that cannot be translated from VAX to Alpha, 395
transfer of control to, 360–364
translating image names, 328
uninstalled
activating, 303
virtual address space, 302–303
**IMAGE_TYPE linker option**
identifying image as CLI, 313
**IMCB (image control block)**
characteristics and use, 317–319
done list
location of, 317
moving IMCBs to, 335
field definitions, 317–319
layout, 318 (fig.)
lookaside list, location of, 317
manipulating, example, 343
queue listhead, 317
released by MMG$IMGRESET, 374
SHL field that specifies the address of, 352
stack, 341 (fig.)
work list
location of, 317
moving IMCBs from, 335
processing, 329
**IMCB$L_ACTIVE_SONS field**
role during initialization order computation, 342–343

**IMG$DO_WORK_LIST routine (IMGMAPISD module)**
initialization order computation, control flow, 341–343
main loop, image activator, 329
**IMG$GL_IMAGE_LIST cell**
system IMCB queue listhead, 359
**IMG$GL_IMCB_LIST cell**
IMCB queue listhead, 317
**IMG$PRVSHRIMG routine (SYSIMGFIX module)**
activating a privileged shareable image, operations, 347
**IMG$_IMAGE_NOT_FOUND error status**
returned by EXE$IMGFIX, 360
**IMG$_NOTNATIVE error status**
returned by EXE$IMGACT, 329
**$IMGACT (Image Activator system service).** *See also* image activation; image rundown.
activating
shareable image, control flow, 335–338
shareable image, example, 343–344
simple main image, control flow, 326–335
arguments, 327 (table)
buffers, IMCB field that specifies, 319
end processing, control flow, 338–340
initialization order computation, 340–343
lock use by, characteristics and use of each lock, 495–497
overview, 325
**$IMGFIX (Address Relocation Fixup system service).** *See also* address fixup; address relocation; fixup section; *Consult* OpenVMS Linker Utility Manual.
operations, 359–360
overview, 325
processing IMCBs, operations, 325
**IMGIOCNT parameter (SYSGEN)**
size of image I/O segment, 338
**IMGMAPISD module.** *See* IMG$DO_WORK_LIST routine.
**IMGREG_PAGES parameter (SYSGEN)**
size of P1 space for nonresident image sections, 359
**$IMGSTA (Image Startup system service)**
operations, 362–363
**implicit affinity.** *See* CPU affinity—implicit.
**IMSEM$IL_FLAGS field**
definition and use, 186
**IMSEM$IL_HISTORY_IDX field**
definition and use, 187
**IMSEM$V_CHECK_UPCALL bit (IMSEM$IL_FLAGS field)**
definition and use, 187
**IMSEM$V_EAST_PENDING bit (IMSEM$IL_FLAGS field)**
definition and use, 186, 197
use by AST exit code, 200
**IMSEM$V_KAST_PENDING bit (IMSEM$IL_FLAGS field)**
definition and use, 186, 197
use by AST exit code, 200
**IMSEM$V_RS_WAITRS bit (IMSEM$IL_FLAGS field)**
definition and use, 187
use by AST exit code, 200
**IMSEM$V_TM_WAITRS bit (IMSEM$IL_FLAGS field)**
definition and use, 187
**IMSEM$W_OWNER field**
definition and use, 186
**IMSEM$W_OWNER_DEPTH field**
definition and use, 186
**IMSEM_ACQUIRE routine (SYSTEM_SERVICE_DISPATCHER module)**
acquiring inner mode semaphore, control flow, 188–189
monitoring version, 189
waiting for inner mode semaphore, control flow, 190
**IMSEM_RELEASE routine (SYSTEM_SERVICE_DISPATCHER module)**
releasing inner mode semaphore, control flow, 193–194
releasing user mode threads, control flow, 194
**INCONMMGST bugcheck.** *See also* bugchecks.
generated by
DELETE_IN_SYS_CONTEXT, 429
**INCONSTATE bugcheck.** *See* bugchecks.
**INCON_SCHED bugcheck.** *See also* bugchecks.
generated by
EXE$CALLBACK_JACKET, 208
EXE$INFORM_TM_CALL, 207
SCH$FIND_NEXT_PROC, 72
**INCON_SHELL bugcheck**
generated by SWP$SHELL_INIT, 288
**Individual Departure lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 498
**Individual Index lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 497–498
**Individual Link Registration lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 498–499
**initialization procedures**
main image, 361
shareable image, 319, 362
translated image, 404, 406
**INIT_INTERACTIVE routine ([LOGIN]INTERACT module)**
logging in interactive processes, control flow, 445–446

**$INIT_MULTITHREAD system service**
control flow, 164–165
initializing DECthreads data, 212
**inner access mode**
term definition, xiii
**inner mode semaphore**. *See also* IMSEM_ACQUIRE routine; IMSEM_RELEASE routine.
acquiring, 187–189
by AST procedure, 185
by system service, 185
when delivering outer mode AST with a PKAST, 201
acquisition flow, 189 (fig.)
characteristics and field definitions, 186–187
contained in FRED, 161
effect on AST delivery, 186
layout, 186 (fig.)
overview, 185
relation to INNER_MODE wait, 40
releasing, 193–194
ring buffer
definition and use, 187
displaying contents of, 187
role in servicing non-thread-safe AST process queues, 199
use
during kernel thread deletion, 176–177
during process suspension, 247
by EXE$SOFT_SUSPND_INT, 251
wait flow, 191 (fig.)
**input/output**. *See* I/O.
**Install lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 496–497
**Install utility**
address relocations and fixups, actions, 310
/HEADER_RESIDENT qualifier, 304
images installed /RESIDENT, actions, 307
known image database created and managed by, 304
lock use by, characteristics and use of each lock, 495–497
naming global sections, 304
/OPEN qualifier, 304
overriding base address assignments, 358
P1 space for nonresident image sections managed by, 359
preactivating images, 359
/PRIVILEGED qualifier, 304
/PROTECTED qualifier, 305
qualifiers, 304–306
/RESIDENT qualifier, 305
/SECTION_BINDING qualifier, 305, 306
/SHARE qualifier, 305
subset fixup sections created by, 346
**instruction emulation**. *See also* instructions.
VAX, for translated images, 386–391
**instruction granularity bitmap**
description, 391
use, 414
**instructions**. *See also* Alpha architecture.
LDx_L. *See* load-locked/store-conditional mechanism.
MB
queuing an AST to a multithreaded process, 198
releasing inner mode semaphore, 193
use at event flag clearing, 102
use at event flag reading, 102
use at event flag setting, 100, 211
use by event flag wait services, 91, 92
use by translated images, 388
RC, characteristics and use, 389
RS, characteristics and use, 389
STx_C. *See* load-locked/store-conditional mechanism.
TRAPB
use by $NATIVE_TO_TRANSLATED system service, 402
use in translated images, 387
**inswap**. *See also* swapper.
term definition, 6
**integer registers**
as part of hardware context, 3, 20
**interactive processes**
creating, 440 (fig.)
arguments passed to $CREPRC that result in, 440 (table)
by job controller, 439
DCL spawning of, 461–465
limit, 448
LOGINOUT operations, control flow, 443–451
logout, LOGINOUT control flow, 468–470
subprocesses, DCL commands that create and connect with, 461
**INTERACTIVE_VALIDATION routine ([LOGIN]INTERACT module)**
logging in interactive processes, control flow, 445–446
**internal process ID**. *See* IPID.
**interoperability**
native and translated images, 396
**interoperable images**
image activator operations, 347
**interprocess communication**
common event flags use for, 259
global sections use for, 261
logical names use for, 260–261
mailbox use for, 260
mechanisms
characteristics and use, 259–261

**interprocess communication** *(continued)*
mechanisms *(continued)*
list of available, 223
synchronization
through common event flags, 104
through lock management system services, 259–260
**interprocessor interrupt work requests**
translation buffer invalidation, 182
**interrupt priority level**. *See* IPL.
**interval timer**
interrupt service routine
role in quantum expiration, 209
use of capabilities, 44
**INVEXCEPTN bugcheck**. *See* bugchecks.
**INVPTBR bugcheck**. *See* bugchecks.
**INVSCBB bugcheck**. *See* bugchecks.
**IOCSLAST_CHAN routine (IOSUBNPAG module)**
explicit affinity for primary processor, 44
**IOCIOPOST module**. *See* PAGIO_DONE routine.
**IOSB (I/O status block)**. *Consult* OpenVMS I/O User's Reference Volume; OpenVMS System Services Reference Manual.
synchronous system services return path handling of, 94–97
**IOSEGMENT linker option**
specifying size of image I/O segment, 313, 338
**IOSUBNPAG module**
IOCSLAST_CHAN. *See* IOCSLAST_CHAN routine.
**IOTA parameter (SYSGEN)**
deducted from KTB$L_QUANT, 56
SCHSWAIT_PROC use, 49
**IPID (internal process identifier)**. *See also* PID.
characteristics and use, 225, 279–280, 282
contents, 154
copying during subprocess creation, 271
creating, for a kernel thread, 154, 171, 281
EPID use compared to, 225
fabricating, 281 (fig.)
LKB field that contains, 118
master process in job tree recorded in CEB, 86
not constant for a user mode thread, 215
not unique across VMScluster nodes, 279
relation to
kernel thread index, 282
PCB, 280
reusable, 154, 281
routines that transform and manipulate, 279–280
sequence number always positive, 282
transforming into EPID, 154
unique for each kernel thread, 154, 280
validating kernel thread IPID, 282
validating process IPID, 282
**IPL (interrupt priority level)**
raising
during image exit, 366
during process rundown, 373
**IPL 2 interrupt**. *See* AST delivery interrupts.
**IPL 3 interrupt**. *See* rescheduling interrupt.
**IPL 7 interrupt**. *See* software timers—interrupt.
**IPL 8**
lock database access synchronized at, 127, 138
**IPL 22 interrupt**. *See* interval timer—interrupt service routine.
**IPL register**. *See* IPL.
**IPL$_LCKMGR**. *See* IPL 8.
**IPL$_RESCHED**. *See* IPL 3 interrupt.
**IPL$_SCHED**. *See* IPL 3 interrupt.
**IPL$_SCS**. *See* IPL 8.
**IPL$_SYNCH**. *See* IPL 8.
**IPL$_TIMER**. *See* IPL 8.
**IPL$_TX_SYNCH**. *See* IPL 8.
**IRP (I/O request packet)**. *See* ACB; SQIO.
**IRP extension**. *See* IRPE.
**IVSSRVRQST bugcheck**. *See* bugchecks.

**jacket procedures**
characteristics and use, 398
**jacketing**
automatic, characteristics and use, 398–401, 399 (fig.)
initiated by the executive, 401
minimal, criteria required for, 399–400
prohibited by the executive, 401
system services, 401–404
through
AMACSEMUL_CALL, 400
BLISCALLG, 400
OTSSCALL_PROC, 400
**JIB (job information block)**
copying during process creation, 266 (fig.)
initializing during process creation, 271–272
KTB address of, 14
layout, 272 (fig.)
obtaining information from, $GETJPI operations, 238
PCB relation with, for several processes in same job, 273 (fig.)
software context contained in, 6
swapper process does not contain, 238
**JIB$L_BYTCNT field**
job quota wait use of, 41
**JIB$L_FLAGS field**
job quota wait use of, 41
**JIB$L_JOBTYPE field**
use by LOGINOUT, 447
**job**
characteristics and use, 2
limit, 448
term definition, 2

**job controller**
accounting message sent at process deletion, 425
lock use by, characteristics and use of each lock, 501–506
New Job Controller lock use by, 503–504
Node Identifier lock use by, 503
process creation by, 439–441
sharing resources in VMScluster systems, 472
**job information block**. *See* JIB.
**job quota**. *See also* JIBSL_FLAGS field.
charging locks against, 129
depleted
miscellaneous wait triggered by, 40
transition states triggered by, 40–41
recorded in JIB, 2
waiting for, context, 54
**JOB_CONTROL process**. *See* job controller.

**kernel**. *See also* memory management; scheduling.
components, xi
**kernel mode**
innermost access mode, xiii
stack pointer, as part of hardware context, 20
suspension
AST queuing during, 249
characteristics, 246, 248
SUSPND_INTERIM_AST actions, 249
**kernel mode stack**. *See* kernel stack.
**kernel request packet**. *See* KRP.
**kernel stack**
canonical. *See* canonical kernel stack.
contents at process creation, 284 (fig.)
hardware context saved on, 21
pages
locked in process working set list, 161
protection, 161
**kernel thread block**. *See* KTB.
**kernel thread states**. *See also* CEF; COLPG; COM; COMO; CUR; event flag wait state; HIB; HIBO; INIT processor state; LEF; LEFO; MWAIT; PFW; SUSP; SUSPO; wait states.
display of resource wait, 37
KTB field that specifies, 13
queues
characteristics and use, 15–17
computable, 16 (fig.)
KTB fields that link, 13
scheduler database component, 11
wait, characteristics and use, 15–17
scheduling
symbolic name for, format, 32
as significant scheduling characteristic, 9
symbolic names and values, 34 (table)
transitions, 33 (fig.)
control flow, 60–63
from CUR state, control flow, 46–52, 48 (fig.), 51 (fig.)
**kernel threads**. *See also* multithreaded processes; processes; user mode threads.
access modes, 3
characteristics most important for scheduling, 9
characteristics shared by, 178
class-scheduled, transforming into normally scheduled, 78
common event flags shared by, 81
converting locks, 116
creating. *See also* process creation.
main steps, 170 (fig.)
operations, 169–173
data structures, relations among, 155 (fig.)
deleting. *See also* process deletion.
in context of caller, 174–175
in context of kernel thread being deleted, 175–177
context switching at, 76
with explicit affinity, 44
main steps, 174 (fig.)
makes kernel thread computable, 36
operations, 173–178
in system context, 177–178
dequeuing locks, 116
dormant
determination, 67
effect on pixscan history, 67
execution, interrupted by, 31
$GETJPI wildcarding through, 242–243
I/O-bound, quantum-end mechanism for, 56
ID, relation to IPID, 282
identifying, 154, 225, 279–283
through HWPCB address, 161 (ex.)
independent of others in same process, 180
initial
characteristics and use, 226
in COMO state, 34
created at process creation, 169
$EXIT executes in, 436
FRED location, 156
ID, 154
KTB address, 156
KTB equivalent to process PCB, 237
multithreading termination, 167
PCB and KTB, 154
PKTA location, 162
queuing AST to, during forced exit, 253
initializing, 361
initiating execution in new, 173
life cycle, 169
local event flags shared by, 81
locating with KTB vector, 236
lock requests issued by, 108
lock use by, 259

**kernel threads** *(continued)*
maximum number per process, SYSGEN parameter that specifies, 153
multiplexing user mode threads in, 152
normal
preempting, 31
priority range, 31
scheduling, 31
scheduling, distinguished from real-time kernel threads, 30
placing
back into execution, nonprivileged hardware context, 24
into hibernation, 244–245
into event flag wait state, 90
preemption, rules for choosing a new CUR, 29
priorities in scheduling, 178–180
priority. *See also* priorities.
altering, 255–256
set at kernel thread creation, 171
private area. *See also* PKTA.
calculating size of, 162
contents, 161
defined by SHELLxK module, 162
process name
can be changed by, 257
shared by, 257
quantum. *See* quantum.
quiescent
receiving inner mode ASTs while suspended, 201
waiting in SUSP state until $SINGLE_THREAD is requested, 201
real-time
priority handling, 63
priority range, 30
scheduling, distinguished from normal kernel threads, 30
removing from hibernation, 245–246
rescheduling current, 256–257
resuming, 252
scheduling, 8
example, 180
example flow, 179 (fig.)
scheduling entity in OpenVMS, 178
scheduling models, 152
selecting for execution, 9
servicing non-thread-safe AST process queues, 199
state change, upcall performed as a result of, 204
status, 14
suspending, 246–252
synchronizing
with system services, event flag use for, 81
term definition, 2, 151

**KFD (known file directory)**
field definitions, 321
KFEs and, 322 (fig.)
layout, 321 (fig.)
**KFE (known file entry)**
characteristics and use, 304, 319–321
field definitions, 320–322
global section name descriptor initialized with address of, 328
hash table, locating KFEs, 322
IMCB field that specifies, 319
KFD and, 322 (fig.)
layout, 320 (fig.)
list, characteristics and use, 496–497
lock, characteristics and use, 495
**KFE lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 495
**KFE$Q_PROCPRIV field**
process privilege mask, use and routines that manipulate it, 380 (table)
**KFERES (known file entry resident section descriptor)**
characteristics and use, 322–325
deallocated by MMGSIMGRESET, 374
field definitions, 324
IMCB field that specifies, 319
KFE field that specifies, 321
layout, 323 (fig.)
**KFPB (known file pointer block)**
field definitions, 322
layout, 323 (fig.)
**KFRH (known file resident image header)**
layout, 321 (fig.)
**known file database**. *See* known image database.
**known file directory**. *See* KFD.
**known file entry**. *See* KFE.
**known file entry resident section descriptor**. *See* KFERES.
**known file pointer block**. *See* KFPB.
**known file resident image header**. *See* KFRH.
**known image database**
characteristics and use, 304
**known images**
characteristics and use, 304
image activator operations, 345
**$KRNDWN (Image Rundown system service)**. *See also* image rundown.
control flow, 371–377
rundown routines, invoked by, 371
**KRNLSTAKNV bugcheck**. *See* bugchecks.
**KRP (kernel request packet)**
allocated by EXE$CREPRC during process creation, 268
**KRP lookaside list**
initialized by EXE$PROCSTRT, 292

**KRPEMPTY bugcheck**. *See* bugchecks.
**KSTACKPAGES parameter (SYSGEN)**
limiting resource tree maximum depth with, 145
P1 space initialization based on, 287
size of kernel thread kernel stack, 162
use by EXE$INIT_MULTITHREAD, 164
**KTB (kernel thread block)**
addresses stored in KTB vector, 156
allocated from nonpaged pool, 156
characteristics and use, 6, 11
contents, 226
copying during process creation, 266(fig.)
event flags, fields, 86–87, 87(fig.)
field definitions, 154–156
field that contains PKTA address, 161
fields related to multithreading, 197
initial kernel thread
overlays PCB, 154
initializing during process creation, 274–275
kernel thread state queue fields, 13
layout, 157(fig.)
locating with thread ID, 282
location of, 13
obtaining information from, $GETJPI operations, 238
overlays PCB, 11
PHD address contained in, 11
PKTA field that contains address of, 162
queues, listheads, 79
scheduler database component, 11
scheduling-related fields, 13–15, 14(fig.)
**KTB vector**
characteristics and use, 282
contains all KTB addresses, 156
locating process kernel threads, 236
for multithreaded process, 159(fig.)
for single-threaded process, 160(fig.)
**KTB$L_AFFINITIES field**
definition and use, 43, 44
**KTB$L_AFFINITY_SKIP field**
definition and use, 45
**KTB$L_ASTQxL_E field**
definition and use, 197
**KTB$L_ASTQxL_K field**
definition and use, 197
**KTB$L_ASTQxL_S field**
definition and use, 197
**KTB$L_ASTQxL_SPK field**
definition and use, 197
**KTB$L_ASTQxL_U field**
definition and use, 197
**KTB$L_AST_PENDING field**
definition and use, 197
**KTB$L_AUTHPRI field**
definition and use, 28
**KTB$L_BIAS_CELL field**
definition and use, 45, 275
**KTB$L_CAPABILITIES field**
definition and use, 42, 44, 45
**KTB$L_CAPABILITIES_SEQ field**
definition and use, 43
**KTB$L_CPU_ID field**
definition and use, 15
**KTB$L_CURRENT_AFFINITY field**
definition and use, 43
reset at image rundown, 44
use by SCH$REPORT_EVENT, 62
**KTB$L_EFWM field**
contents for a kernel thread in event flag wait state, 93
contents for inner mode semaphore wait, 192
definition and use, 14, 87
identifies MWAIT entity, 37
JIB address of kernel thread in job quota wait, 41
mutex address of kernel thread in mutex wait, 40
resource number for process in resource wait, 87
**KTB$L_EPID field**
EPID location, 225
**KTB$L_FRED field**
definition and use, 14
**KTB$L_JIB field**
definition and use, 14
**KTB$L_MTXCNT field**
definition and use, 15
**KTB$L_ONQTIME field**
definition and use, 15
use by
quantum expiration, 56
SCH$PIXSCAN, 67
SCH$REPORT_EVENT, 62
**KTB$L_PERMANENT_AFFINITIES field**
definition and use, 43
**KTB$L_PERMANENT_CAPABILITIES field**
definition and use, 42
**KTB$L_PER_KT_AREA field**
address of permanent PKTA, 161, 275
**KTB$L_PHD field**
definition and use, 14
**KTB$L_PID field**
IPID location, 225
**KTB$L_PRI field**
definition and use, 28
kernel thread priority, 269
new priority after system event, 63
**KTB$L_PRIB field**
definition and use, 28
determining priority with, 63
kernel thread priority, 269
**KTB$L_PRIBSAV field**
definition and use, 28

**KTB$L_PRISAV field**
definition and use, 28
**KTB$L_QUANT field**
interval timer interrupt service routine use of, 55
quantum expiration use of, 56
scheduling-related PHD field, 15
**KTB$L_SAVED_AFFINITIES field**
definition and use, 44
**KTB$L_SAVED_CAPABILITIES field**
definition and use, 44
**KTB$L_SQBL field**
definition and use, 13
**KTB$L_SQFL field**
definition and use, 13
**KTB$L_STATE field**
definition and use, 13
updated by
SCH$FIND_NEXT_PROC, 72
SCH$STATE_TO_COM, 71
use by SCH$REPORT_EVENT, 62
wait queue values, 16
**KTB$L_STS field**
definition and use, 87, 156, 257
flags, 14
quantum expiration use of, 56
**KTB$L_STS2 field**
definition and use, 156
flags, 14
**KTB$L_WAITIME field**
definition and use, 15
EXE$GL_ABSTIM_TICS contents recorded in, 49
use by SCH$REPORT_EVENT, 62
**KTB$L_WEFC field**
definition and use, 87
**KTB$Q_PHYPCB field**
definition and use, 14
physical address of HWPCB, 22
**KTB$V_INQUAN bit (KTB$L_STS field)**
quantum expiration use of, 56
**KTB$V_IN_TBS_STATE bit (KTB$L_STS2 field)**
definition and use, 78
**KTB$V_QUANTUM_RESCHED bit (KTB$L_STS2 field)**
definition and use, 57, 79
**KTB$V_RES bit (KTB$L_STS field)**
flag for process residence state, 32
scheduling significance of, 14
**KTB$V_SCHED_CONTEXT_SAVED bit (KTB$L_FLAGS field)**
definition and use, 44
**KTB$V_WALL bit (KTB$L_STS field)**
definition and use, 14, 87
**L1PT (level 1 page table)**
initialized by SWPSSHELL_INIT, control flow, 286
**L2PT (level 2 page table)**
initialized by SWPSSHELL_INIT, control flow, 287
**L3PT (level 3 page table)**
initialized by SWPSSHELL_INIT, control flow, 287
**Label lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 476–477
**last chance condition handlers**
user mode, established by DCL, 458
**LCK$GL_HASHTBL cell**
resource hash table address located in, 113
**LCK$GL_IDTBL cell**
pointer to lock ID table, 121
**LCK$GL_LKB_HEAD cell**
LKB cache, 120
**LCK$GL_LKIDCNT cell**
total number of lock IDs, 121
**LCK$GL_LKIDFREE cell**
number of free lock IDs, 121
**LCK$GL_LKNXTID cell**
lock ID table first entry index, 121
**LCK$GL_LSTFREE cell**
index of last free lock ID, 121, 122
**LCK$GL_MAXID cell**
lock ID table last entry index, 121
**LCK$GL_NXTID cell**
lock ID obtained from, 122
**LCK$GL_RRSFL cell**
listhead for root resource list for local system, 110, 130
**LCK$GL_RSB_HEAD cell**
location of RSB cache, 113
**LCK$GL_TIMOUTQ cell**
lock timeout queue listhead, 142
**LCK$GL_WAITTIME cell**
nonzero when deadlock wait enabled, 130, 133, 142
**LCK$GRANT_LOCK routine (SYSENQDEQ module)**
control flow, 134–135
**LCK$SEARCHDLCK routine (DEADLOCK module)**
operations, 143
verifying existence of multiple resource deadlock, 144
**LCK$SRCH_RESDLCK routine (DEADLOCK module)**
operations, 144–146
searching for multiple resource deadlock, example, 147–148
stack frame built for, 145 (fig.)

**LCK$V_CVTSYS flag**
use in dissociating a lock, 472
**LCK$V_SYSTEM flag**
use in VMScluster systems, 472
**LCKMGR spinlock**
access to lock database synchronized by, 127
held during dispatch to system-owned locks blocking AST routine, 136
use by
EXESDEQ, 138
**LDx_L (load-locked) instruction**. *See* load-locked/store-conditional mechanism.
characteristics and use, 98
context for, 53
placing kernel threads into, 93, 97, 210
removing kernel threads from, 101
transitions
from LEF to COM, 35
to LEF from CUR, 35
**LEFO (local event flag outswapped wait state)**. *See also* event flag wait state; event flags; kernel thread states; LEF; local event flags; wait states.
characteristics and use, 98
removing kernel threads from, 101
transitions
from LEFO to COMO, 35
to LEFO from LEF, 35
**level 1 page table**. *See* L1PT.
**level 2 page table**. *See* L2PT.
**level 3 page table**. *See* L3PT.
**LGIS_INVINPUT error status**
signaled by LOGINOUT, 445
**LGI_CALLOUTS parameter (SYSGEN)**
enables LOGINOUT installation callouts, 443
**LIB$ATTACH routine (Run-Time Library)**
ATTACH command functions available to images through, 461
**LIB$FIND_IMAGE_SYMBOL routine (Run-Time Library)**
procedure that initiates merged image activation, 344
**LIB$P1_MERGE routine ([VMSLIB]LIBMERGE module)**
operations, 448
**LIB$PAUSE routine (Run-Time Library)**
interrupting images, program control of, 460
**LIB$SPAWN routine (Run-Time Library)**
SPAWN command functions available to images through, 461
**[LIBOTS]OTS$CALL_PROC_ALPHA module**. *See* OTS$CALL_PROC routine.
**linkage pair**
address fixups, 355 (fig.)
SIMGFIX operations, 356
address reference, requiring adjustment, 348
with procedure signature blocks
address fixups, 357 (fig.)
address fixups, SIMGFIX operations, 356–357
autojacketing outbound calls, 399, 399 (fig.)
with procedure signature blocks, requiring adjustment, 349
term definition, 348
**linkage section**
characteristics and use, 300
created as writable copy-on-reference section, 310
effect of relocations and fixups on, 306
image locations requiring relocation contained in, 352
most address relocations and fixups apply to, 359
when shareable, 309, 359
**linker options**
BUILD_IDENT, identifying build class of image, 315
CLUSTER, effect on pagelet fault cluster factor for sections, 316
file, contents, 310
GSMATCH, specifying global section major and minor IDs, 314
IDENTIFICATION, specifying image identification, 315
IMAGE_TYPE, identifying image as CLI, 313
IOSEGMENT, specifying size of image I/O segment, 313, 338
NAME, specifying image name, 314
PROTECT, protecting image sections, 302
STACK, specifying size of user stack, 339
**LKB (lock block)**
ACB contained in, 118
characteristics and use, 116–120
distribution of, in a VMScluster system, 122
field definitions, 116–123
layout, 117 (fig.)
linking into lock timeout queue, 142 (fig.)
local copy, 123
locating, 120, 122
master copy, 123
overview, 108
process copy, 123
removing from resource queue during dequeuing operations, 138
types, 123
distinguished by LKB and RSB contents, 123
used as ACB for AST delivery, 128
**LKB$GL_NXTID cell**
listhead for unused lock ID table entries, 121
**LKB$L_DUETIME field**
synonym for LKB$L_KAST, storing lock request timeout time in, 142

**LKB$L_KAST field**
synonym for LKB$L_DUETIME, storing lock request timeout time in, 142
**LKB$L_LCKCTX field**
fork process–owned lock use, 137
**LKB$L_PID field**
fork process–owned lock use, 137
**LKSB (lock status block)**
address and condition value, LKB field that contains, 119
synchronous system services return path handling of, 94–97
**LNMSAQ_MUTEX cell**
locked by EXE$PROCSTRT, 294
**LNM$PROCESS_DIRECTORY directory**
initialized by EXE$PROCSTRT, 294
**LNMHSH (logical name hash table)**
size of process table, 293
**load-locked/store-conditional mechanism**
acquiring inner mode semaphore, 188
common event flags set and cleared by, 86, 101, 102
during event flag setting, 211
local event flags set and cleared by, 83, 100, 102
queuing an AST to a multithreaded process, 198
releasing inner mode semaphore, 193, 194, 195
waiting for inner mode semaphore, 192
**loadable executive images**. *See* executive images.
**local event flag outswapped state**. *See* LEFO.
**local event flags**. *See also* event flags; LEFO.
characteristics and use, 83
clusters 0 and 1, location, 83
load-locked/store-conditional mechanism use of, 83
located in PHD, 83
numbers
available to processes, 81
reserved for system use, 83
SCHSPOSTEF actions when setting, 100–101
shared among process kernel threads, 81
systemwide wait state queues, characteristics and use, 98
**lock access mode**
how determined, 118
**lock block**. *See* LKB.
**lock ID table**
characteristics and use, 120–122
expanding, 121–122
location, 121
structure, 120(fig.)
SYSGEN parameters that control size of, 121
**lock management system**
data structures, characteristics and field definitions, 109–125
lock database
accessing, 122
organization, in a VMScluster system, 122–125
relations in the, 122
overview, 107–109
system services
interprocess communication mechanism, 259–260
operations, 126–140
**lock modes**
characteristics and use, 117(table)
Device lock use of, 475
effect on lock requests, 118, 125
RMS use of, 487
term definition, 471
**lock requests**
effect of lock modes on, 118, 125
granting, by lock management system services, 108
issued from context of requesting kernel thread, 108
for nonlocal resource, directory mode operations, 124
null mode, 131
rules for granting, 125
states, 108
waiting, rules for granting, 126
**lock status block**. *See* LKSB.
**lock timeout queue**
deadlock handling use of, 142
location, 142
**LOCKDIRWT parameter (SYSGEN)**
effect on resource mastership, 124–125
**LOCKIDTBL parameter (SYSGEN)**
lock ID table size controlled by, 121, 122
**LOCKIDTBL_MAX parameter (SYSGEN)**
lock ID table size controlled by, 121, 122
**LOCKMGRERR bugcheck**
generated by the executive, 122
**locks (lock management system)**. *See also* LKB; locks used by OpenVMS Alpha components; protection; resources; RSB; synchronization.
blocking a specific lock, 140
blocking ASTs, use by lock management system, 109, 134, 136, 139, 142
blocking condition, lock conversion use of, 131
compatibility, 107, 118
completion ASTs, use by lock management system, 130, 134, 140
condition of current, LKB that specifies, 118
conversion grant mode, 132
compared with grant mode, 132–133
distinguished from group grant mode, 132
conversion requests
determining when system-owned, 136
handling incompatible, 133
to other modes, 131–133
process-owned to system-owned, 136

rules for granting, 125–126
converting, by kernel threads, 116
deadlock handling. *See* deadlocks.
dequeuing
$DEQ operations, 137
by kernel threads, 116
lock ID role, 121
fork process–owned, 137
grant mode, 131
granting
circumstances under which routines request, 133
determining when system-owned, 136
rules for, 125
group grant mode, 131
ID. *See also* lock ID table.
LKB field that contains, 119
validating, 121
index, 120
information about obtaining
$GETLKI operations, 139
restrictions on, 139
locating, 121
maximum number allowed, SYSGEN parameter that specifies, 121
modes, role in lock conversion, 131–132
multiple, dequeuing, 137
overview, 107–108
owned by processes, not kernel threads, 116
reference count, decrementing when sublock is dequeued, 138
request mode, 131
requests for
canceling ungranted, 137
precedence handling, 118
restricting participation in deadlock searches, methods for, 142
sublock relations, 119 (fig.)
system services used with, 108
system-owned
characteristics and use, 135–137, 472
distinguished from systemwide resource, 135
how access modes determined, 136
OpenVMS components that use, 136
process must be associated with system domain to access, 115
timeout queue, linking locks into, 118
use
by kernel threads, 259
during process rundown, 373
value block, characteristics and use, 473
wait queue, placing into, 130
**locks used by OpenVMS Alpha components**
Alias, VMScluster use of, 497–499
Bucket, 489–491
Cache, 482–485
Device, 474–476
Disk Quota Cache, 484
$DISMOU use of, 477–478
Dismount, 477–478
Driver-Loading and Autoconfiguration, 507
EOF, 488–489
executive use of, 473–476
Extent Cache, 483–484
File, 486–488
File Access Arbitration, 480–481
File ID Cache, 483
File Serialization, 481–482
file system use of, 478–485
Forms Cache, 505
Global Buffer Backing, 493–494
Global Buffer Master, 492
Global Buffer Section, 492–493
global buffer use of, 491–494
$IMGACT use of, 495–497
Individual Departure, 498
Individual Index, 497–498
Individual Link Registration, 498–499
Install, 496–497
Install utility use of, 495–497
job controller use of, 501–506
KFE, 495
Label, 476–477
Master File Access, 502
Master File Check, 502–503
Master Queue, 506
Master Registration, 497
Modified Proxy, 499–500
Mount Device, 477
$MOUNT use of, 476–477
NETACP use of, 497–501
New Job Controller, 503–504
New Master File, 501
Node Identifier, 503
overview, 471–473
process lock queues, 122
proxy access, networks use of, 499–501
Proxy Function, 500
Proxy Key, 500–501
Queue Definition and Journal Files, 504–505
Queue Manager Alive, 504
queue manager use of, 501–506
Queue ORB, 506
Quota Cache Entry, 484–485
Record, 494–495
RMS use of, 485–495
Set Time, 474
SMISERVER Main, 507
SYSMAN use of, 507
System ID, 473–474
Volume Allocation, 478–479
Volume Blocking, 479–480

**logical name hash table**. *See* LNMHSH.
**logical name tables**
  jobwide
    deleted at top-level process deletion, 424
  process-private
    created by EXE$PROCSTRT, 293–294
**logical name translation block**. *See* LNMX.
**logical names**. *See also* logical name tables; *Consult* OpenVMS User's Manual; OpenVMS Programming Concepts Manual; OpenVMS DCL Dictionary; OpenVMS System Services Reference Manual.
  attributes
    passed at process creation, 274
  executive mode, created by LOGINOUT, 449
  interprocess communication mechanism, 260–261
  spawned subprocess, inherited from creator, 464
**login**
  batch process, 451–452
  disable overridden with OPER privilege, 448
  interactive process, 445–451
  network process, 452
**LOGINOUT.EXE image**
  callout routines, operations, 443
  effect on process capabilities, 439
  first main image to run in process, 434
  logout, control flow, 468–470
  major functions, 442
  mapping CLI into P1 space, 434
  operations
    in batch processes, control flow, 451–452
    in context of newly spawned subprocess, 463
    in interactive processes, control flow, 443–451
    in multiple-image processes, 466
    in network processes, 452
    overview, 442–443
  process attributes reset by, 446–447
  rundown by CLI, 453
**[LOGIN]INTERACT module**. *See* INIT_INTERACTIVE routine; INTERACTIVE_VALIDATION routine.
**[LOGIN]LOGIN module**. *See* LOGOUT routine.
**logout**
  LOGINOUT control flow, 468–470
**LOGOUT routine ([LOGIN]LOGIN module)**
  control flow, 469
**LOG_IO (do logical I/O privilege)**. *See* privileges.
**longword index**
  term definition, xiv
**longwords**
  how represented in figures, xiv, xv
**lookaside lists**. *See* ICB; KRP; pool—nonpaged; PQB.
**macros**
  identifying, xiii
**mailboxes**. *See also* logical names.
  attach request
    for attaching to a process, 462
    for returning interactive control to a DCL process, 465
  communication
    deleting, 465
    with spawned subprocess, 462
    for transmitting process context information to spawned subprocess, 463
    use when spawning subprocess, 464
  full, state transition triggered by, 37
  interprocess communication, 260
  output
    characteristics and use, 462
    write attention AST used as, 462
  processes running on different VMScluster nodes cannot share, 260
  termination
    created by DCL, 462
    deleting, 465
    established at process creation, 274
    message sent to owner process during process deletion, 425 (table)
    sending message to deleted process's creator, 424
    shared by up to four spawned subprocesses, 462
  wait state, characteristics and use, 39
**main image**
  term definition, 302
**Master File Access lock**. *See also* locks used by OpenVMS Alpha components.
  characteristics and use, 502
**Master File Check lock**. *See also* locks used by OpenVMS Alpha components.
  characteristics and use, 502–503
**Master Queue lock**. *See also* locks used by OpenVMS Alpha components.
  characteristics and use, 506
**Master Registration lock**. *See also* locks used by OpenVMS Alpha components.
  characteristics and use, 497
**MAXPROCESSCNT parameter (SYSGEN)**
  effect on EPID layout, 283
  maximum number of processes permitted, 10, 280
**MB instruction**. *See also* memory barriers.
  queuing an AST to a multithreaded process, 198
  releasing inner mode semaphore, 193
  use at
    event flag clearing, 102
    event flag reading, 102
    event flag setting, 100, 211

use by
event flag wait services, 91, 92
translated images, 388
**mechanism array**. *See* condition handlers.
**memory**
access
Alpha compared to VAX, 387–389
atomic move PALcode routines, 388, 389
atomic, characteristics, 388–389
atomic, VEST handling of, 388
how represented in figures, xiv
kernel thread priorities for residence, concepts and mechanisms, 27
mapping registers, 20
order of reads and writes. *See* read-write ordering.
physical. *See* physical memory.
sharing, implications of, 182–183
virtual. *See* virtual memory.
**memory barriers**. *See* MB instruction; read-write ordering.
**memory management**. *See also* address space; global sections; page faults; pool; swapper; virtual memory; working set list.
wait states
characteristics and transitions, 36–37
context for, 53
**message argument vector**. *Consult* OpenVMS System Services Reference Manual.
**message files**. *Consult* OpenVMS Command Definition, Librarian, and Message Utilities Manual; OpenVMS DCL Dictionary.
**message sections**
image activator operations, 337
**MINWSCNT parameter (SYSGEN)**
use during working set initialization, 290
**miscellaneous wait state**. *See* MWAIT.
**MMG spinlock**
use by
DELETE, 426
DELETE_IN_SYS_CONTEXT, 428
EXE$CHK_WAIT_BHVR, 55
EXE$CREPRC, 276
EXE$HIBER_INT, 244
EXE$WAIT, 92
executive, 183
SWP$SHELL_INIT, 286
**MMG$CRETVA_K routine (SYSCREDEL module)**
kernel thread creation, operations, 171
**MMG$IMGRESET routine (PHDUTL module)**
image and process rundown, control flow, 373–375
single-image process, operations, 438
**MMG$PGFLTWAIT routine (PAGEFAULT module)**
waiting a kernel thread, 50
**MMG$RESRCWAIT routine (PAGEFAULT module)**
waiting a kernel thread, 50
**MMG$SWAP_PTBR routine (EXCEPTION module)**
adopting page tables of new process, 286
**MMG$WRTMFYPAG routine (WRTMFYPAG module)**. *See* modified page writer.
**MMG_CTLFLAGS parameter (SYSGEN)**
enabling proactive memory reclamation, 54
**MMG_STD$SWAP_PTBR routine (EXCEPTION module)**. *See* MMG$SWAP_PTBR routine.
**mode of caller system services**. *See* access modes; system services.
**modes**. *See* access modes; change mode; conversion grant mode; executive mode; grant mode; group grant mode; kernel mode; lock modes; special kernel mode AST; supervisor mode; user mode.
**modified page list**
wait state, characteristics and use, 39
**modified page write**
transitions triggered by, 36
**modified page writer**. *See also* MMG$WRTMFYPAG routine.
wait state, characteristics and use, 39
**Modified Proxy lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 499–500
**Modify CPU User Capabilities system service**. *See* SCPU_CAPABILITIES.
**Modify Process Affinity system service**. *See* $PROCESS_AFFINITY.
**Modify Process Implicit Affinity system service**. *See* $SET_IMPLICIT_AFFINITY.
**Modify Process User Capabilities system service**. *See* $PROCESS_CAPABILITIES.
**modular executive**. *See* executive images.
**$MOUNT (Mount Volume system service)**
lock use by, characteristics and use of each lock, 476–477
**Mount Device lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 477
**mount verification**. *Consult* OpenVMS System Manager's Manual.
**Mount Volume system service**. *See* $MOUNT.
**mounted volumes**
jobwide, dismounted at process deletion, 423
**MPW_LOWAITLIMIT parameter (SYSGEN)**
transitions triggered by, 39
**MPW_WAITLIMIT parameter (SYSGEN)**
transitions triggered by, 39
**MTXCNTNONZ bugcheck**. *See* bugchecks.
**multiple active signals**. *See* condition handlers.

**multiple-image processes**. *See* processes—multiple-image.
**multiprocessor systems**. *See* SMP systems.
**MULTITHREAD parameter (SYSGEN)**
effect on upcalls, 216, 217
maximum number of kernel threads, 2
per process, 153
multithreading scenarios based on, 153
use by EXE$INIT_MULTITHREAD, 164
use in calculating kernel thread ID, 281
**multithreaded processes**
AST delivery to, 195, 199–201
AST-related data structures, 196–197
current thread count, 13
event flag setting, 210
when EFN$C_ENF is used, 101
$EXIT operations, 436
image exit, operations, 366
maximum thread count, 12
preventing temporary changes to process working set, 183
queuing ASTs to, 197–198
setting common event flags for, 101
setting local event flags for, 100–101
swapper trimming of, 181
swapping, 181
synchronization within, 183–204
term definition, 2, 151
testing for event completion, 96
wait services actions, 92
**multithreading**
data structures related to, 153–163
initiating, 164–165
main steps, 170(fig.)
many-to-few threading
term definition, 152
many-to-one threading
term definition, 152
MULTITHREAD parameter effect on, 153
parameters that affect, 153
PTHREAD_CONFIG logical name effect on, 153
supported with the executive, 152
system services that initiate and terminate, 163–164
terminating, 167–168
**MUTEX module**. *See* EXE$KERNEL_WAIT_PS routine; SCH$RAVAIL routine; SCH$RESOURCE_WAIT routine; SCH$RESOURCE_WAIT_PS routine; SCH$RESOURCE_WAIT_SETUP routine.
SCH$LOCKR. *See* SCH$LOCKR routine.
SCH$LOCKR_QUAD. *See* SCH$LOCKR_QUAD routine.
SCH$LOCKW. *See* SCH$LOCKW routine.
SCH$LOCKW_QUAD. *See* SCH$LOCKW_QUAD routine.
**MUTEX wait state**. *See also* wait states.
meaning, 37
transition descriptions, 40
**mutexes (mutual exclusion semaphores)**. *See also* CEB mutex; CPU mutex; deadlocks; EXE$GL_CEBMTX cell; EXE$GL_GSDMTX cell; LNM$AQ_MUTEX cell; MTX; MUTEX module; MUTEX wait state; resources; RSDM mutex; scheduling; synchronization.
KTB field that contains number locked by a kernel thread, 15
locked, MWAIT triggered by, 37, 40
locking
saving process priorities during, 28
temporary priority adjustment during, 32
**mutual exclusion semaphores**. *See* mutexes.
**MWAIT (miscellaneous wait state)**. *See also* job quota; kernel thread states; resource wait; wait states.
characteristics and transitions, 37
context for, 54
job quota, transition descriptions, 40–41
KTB field that identifies resource waited for, 14
mutex
transition descriptions, 40
queue, characteristics and use, 37
system resource miscellaneous, transition descriptions, 37–40
transitions to MWAIT from other states, 37
use by EXE$CREPRC, 269

**NAME linker option**
specifying image name, 314
**name space**. *See also* resource domains.
characteristics and use, 113
default is UIC group, 114
extending, 114
system, characteristics and use, 113
**naming conventions**. *See* OpenVMS Alpha listings.
entry points, xii–xiii
**native**
term definition, 384
**native images**
calling translated images, 396
distinguished from translated images, 393
**native procedures**
identifying from procedure value, 397
**$NATIVE_TO_TRANSLATED system service**
characteristics and use, 384
control flow, 402–403
**nested conditions**. *See* condition handlers.
**NET$ACP (network ancillary control process)**
creating network processes, 452
**NETACP (network ancillary control process)**
creating network processes, 452

lock use by, characteristics and use of each lock, 497–501
**NETMBX (create network device privilege)**. *See* privileges.
**networks**
processes, LOGINOUT image operations, 452
proxy access locks, characteristics and use of each lock, 499–501
**New Job Controller lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 503–504
**New Master File lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 501
**NOACNT (suppress accounting messages privilege)**
use by EXE$CREPRC, 268
**NOCALLTRANS bugcheck**. *See also* bugchecks.
fatal, generated by EXE$NATIVE_TO_TRANSLATED, 402
nonfatal, generated by EXE$NATIVE_TO_TRANSLATED, 402
**Node Identifier lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 503
**nonpaged dynamic memory**. *See* pool—nonpaged.
**nonpaged pool**. *See* pool—nonpaged.
**NSA$DEACCESS_AUDIT routine (SECURITY_AUDITING module)**
operations during process deletion, 424
**null frame**. *See* procedures.
**null page**
created during image mapping, 308
**null PCB**
characteristics and use, 10
indicator of unused PCB vector slot, 281
placeholder in PCB vector, 428
placeholder in per-CPU database, 428

**object modules**
names, xii
**object rights block**. *See* ORB.
**one-to-one threading**
term definition, 151
**OPCOM (operator communication) process**. *See* communications; mailboxes.
**[OPCOM]OPCCRASH module**
RWMPE wait state use by, 39
**/OPEN qualifier**
known image installation, 304
**OpenVMS Alpha calling standard**. *See* calling standard.
**OpenVMS Alpha executive**. *See* executive.
**OpenVMS Alpha image header**. *See* EIHD; image header.
**OpenVMS Alpha listings**
names, xii
**OpenVMS Alpha naming conventions**. *See* naming conventions.
**OpenVMS Alpha operating system**. *See also* executive; kernel.
components, xii
emulating VAX environment, 384–391
migrating from OpenVMS VAX, overview, 383–384
**OpenVMS VAX environment**
emulating, 384–391
**OpenVMS VAX image header**. *See* image header.
**OpenVMS VAX operating system**
migrating to OpenVMS Alpha, overview, 383–384
**OPER (perform operator functions privilege)**. *See also* privileges.
use by LOGINOUT, 448
**OPERATOR bugcheck**. *See* bugchecks.
**operator communication process**. *See* OPCOM.
**OPERCRASH bugcheck**. *See* bugchecks.
**ORB (object rights block)**
resource information stored in, 115
used to protect common event flag clusters, 86
**OTSSCALL_PROC routine ([LIBOTS]OTSSCALL_PROC_ALPHA module)**
criteria for generating a call to, 400
operations, 400
**out-of-band ASTs**. *See* AST; attention ASTs.
**outer access mode**
term definition, xiii
**output mailbox**. *See* mailboxes—output.
**outswap**. *See also* swapper.
kernel threads, transitions triggered by, 36
processes
kernel threads enter SUSPO state, 35
term definition, 6

**P0 space**
characteristics and use, 4
deleting during process rundown, 374
**P1 space**
characteristics and use, 4, 6
copying from during process creation, 266 (fig.)
created and initialized by EXE$PROCSTRT, 263
image exit, low-address end that is deleted at, 375 (fig.)
moving process parameters from PQB into, 291 (fig.)
**P2 space**
characteristics and use, 4
**page fault handler**. *See* page faults.
**page fault wait state**. *See* PFW.
**page faults**. *See also* memory management; page fault handler; swapper; working set list.
faulting
modified page out of working set, 39

**page faults** *(continued)*
I/O completion, event causing kernel thread state change, 60
page read completion
transitions triggered by, 36
transitions triggered by, 37, 53
upcalls as a result of, 205
**page lists**. *See* bad page list; free page list; modified page list; zeroed page list.
**page size**
used by linker in creating an image, 303
**page table base register**. *See* PTBR.
**page table entry**. *See* PTE.
**page table space**. *See* PT space.
**page tables**
process
initialization, 283
virtual address space described by hierarchy of, 4
**page transitions**. *See* page faults.
**paged dynamic memory**. *See* pool—paged.
**paged pool**. *See* pool—paged.
**PAGEFAULT module**. *See* MMG$PGFLTWAIT routine; MMG$RESRCWAIT routine.
**PAGEFILE.SYS file**. *See* SYS$SPECIFIC:[SYS-EXE]PAGEFILE.SYS.
**pages**. *See* copy-on-reference pages; global pages; huge pages; null page; page faults; process pages.
null. *See* null page.
**PAGIO_DONE routine (IOCIOPOST module)**
upcalls, operations, 206
**PALcode**. *See* SWPCTX PALcode routine.
**parallel processing**
run-time support, 256
**parent resource**
characteristics and use, 473
**password**
expiration, 450
LOGINOUT prompt, 446
**pause capability**
image interruption operations, 460
**PC (program counter)**
as component of hardware context, 3, 20
translated image use of, 386
**PCB (process control block)**
address in CTL$GL_PCB, 289
buffer object queue initialized by EXECREPRC, 274
characteristics and use, 10
contents, 226
copying during process creation, 266 (fig.)
event flags, fields, 86–87, 87 (fig.)
fields related to multithreading, 197
initialized during process creation, 274–275
JIB relation with, for several processes in same job, 273 (fig.)
layout, 157 (fig.)
locating with IPID, 282
lock queue initialized by EXECREPRC, 274
null, characteristics and use, 10
obtaining information from, $GETJPI operations, 238
overlaid by
KTB, 11
KTB of initial kernel thread, 154
process identification-related fields, 279
process, equivalent to initial kernel thread KTB, 237
relation to IPID, 280
scheduler database component, 10
scheduling-related fields, 11–13, 12 (fig.)
software context contained in, 5–6
status flags defined at process creation, 270 (table)
system. *See* system PCB.
use as initial kernel thread KTB, 269
vector scans, context block fields that track local, 232
**PCB vector**
addresses of PCBs, contained in, 10, 280
contents, 10 (fig.)
scanned by
EXE$CREPRC, 280
EXE$PSCAN_NEXT_IPID, 236
sized by MAXPROCESSCNT, 280
**PCB$L_ACC_WAITIME field**
accumulated wait time, 13, 55
use by
EXE$CHK_WAIT_BHVR, 55
SCH$REPORT_EVENT, 62
**PCB$L_ACTIVE_CPUS field**
CPUs on which threads are active, 182
definition and use, 12
updated by SCH$INTERRUPT, 73
**PCB$L_ARB field**
definition and use, 271
**PCB$L_AUTHPRI field**
use by LOGINOUT, 447
**PCB$L_CLASS_COUNT field**
definition and use, 13
**PCB$L_CLASS_QUANT field**
definition and use, 78
**PCB$L_CPUTIME_REF field**
definition and use, 13
**PCB$L_DEFPROT field**
use by LOGINOUT, 447
**PCB$L_EFC2P field**
definition and use, 85, 86
**PCB$L_EFC3P field**
definition and use, 85, 86
**PCB$L_EOWNER field**
definition and use, 271, 279

**PCB$L_EPID field**
definition and use, 279
EPID location, 225
**PCB$L_FREWSLE_PARAM field**
temporary use, 276
**PCB$L_IM_ASTQxL_E field**
definition and use, 197
**PCB$L_IM_ASTQxL_K field**
definition and use, 197
**PCB$L_IM_ASTQxL_SPK field**
definition and use, 197
**PCB$L_INITIAL_KTB field**
definition and use, 156, 275
**PCB$L_JIB field**
definition and use, 272
**PCB$L_KTBVEC field**
addresses of process KTB vector, 156
definition and use, 13, 275
**PCB$L_KT_COUNT field**
definition and use, 13, 275
**PCB$L_KT_HIGH field**
definition and use, 275
number of highest KTB vector entry, 156
**PCB$L_LEFC_0_SWAPPED field**
definition and use, 86
**PCB$L_LEFC_1_SWAPPED field**
definition and use, 86
**PCB$L_LOCKQBL field**
definition and use, 122
**PCB$L_LOCKQFL field**
definition and use, 122
**PCB$L_MULTITHREAD field**
definition and use, 12
initialized by EXE$INIT_MULTTTHREAD, 165
**PCB$L_NOAUDIT field**
initialized during process creation, 275
**PCB$L_OWNER field**
definition and use, 271, 279
**PCB$L_PID field**
definition and use, 279
IPID location, 225
**PCB$L_PIXHIST field**
definition and use, 67
quantum expiration update, 56
**PCB$L_POSTEF_ACB field**
definition and use, 87
**PCB$L_PPGCNT field**
definition and use, 275
**PCB$L_PQB field**
address of PQB, 276
**PCB$L_PRCCNT field**
definition and use, 272
**PCB$L_PRVCPU field**
definition and use, 25, 73
**PCB$L_STS field**
definition and use, 13, 156, 257
flags, 258
**PCB$L_STS2 field**
definition and use, 13, 156
use by class scheduling, 78
**PCB$L_WSSWP field**
identifying a new process with, 285
**PCB$Q_KEEP_IN_WS field**
characteristics and use, 183
definition and use, 275
use by EXE$SYNCH_INT, 97
**PCB$Q_KEEP_IN_WS2 field**
characteristics and use, 183
definition and use, 275
**PCB$Q_LEFC_SWAPPED field**
definition and use, 86
**PCB$Q_ONCPUCNT field**
definition and use, 12, 73
**PCB$Q_POSTEF field**
definition and use, 87
**PCB$Q_PRIV field**
process privilege mask, use and routines that manipulate it, 380(table)
**PCB$Q_PRVASN field**
definition and use, 25, 73
**PCB$Q_PRVASNSEQ field**
definition and use, 25, 73
**PCB$T_LNAME field**
definition and use, 275
**PCB$T_TERMINAL field**
definition and use, 445
use by LOGINOUT, 447
**PCB$V_CLASS_SCHEDULED bit (PCB$L_STS2 field)**
definition and use, 78
**PCB$V_CLASS_SUPPLIED bit (PCB$L_STS2 field)**
definition and use, 78
**PCB$V_DELPEN bit (PCB$L_STS field)**
marking process for deletion, 420
**PCB$V_DISAWS bit (PCB$L_STS field)**
definition and use, 13
**PCB$V_ERDACT bit (PCB$L_STS field)**
characteristics and use, 379
role in process deletion, 420
set to indicate executive mode rundown, 421
tested during process deletion, 420
use by EXIT_CALLBACK, 438
**PCB$V_NODELET bit (PCB$L_STS field)**
set to prevent process deletion, 419
**PCB$V_NOTIFY bit (PCB$L_STS2 field)**
role in process deletion, 424
**PCB$V_NOUNSHELVE bit (PCB$L_STS field)**
definition and use, 271
**PCB$V_PHDRES bit (PCB$L_STS field)**
definition and use, 290
**PCB$V_RES bit (PCB$L_STS field)**
definition and use, 13

**PCB$V_RESPEN bit (PCB$L_STS field)**
set to resume suspended process, 420
**PCB$V_RWAST bit (PCB$L_STS2 field)**
definition and use, 197
**PCB$V_SECAUDIT bit (PCB$L_STS field)**
definition and use, 270
**PCB$V_SHELVING_RESERVE bit (PCB$L_STS field)**
definition and use, 271
**PCB$V_SSRWAIT bit (PCB$L_STS field)**
disabling resource waits with, 40
role in process deletion, 421
**PCB$V_TCB bit (PCB$L_STS2 field)**
definition and use, 275
**PCB$V_WINDFALL bit (PCB$L_STS2 field)**
definition and use, 78
**PCB$W_CLASS_EXTRA_TICKS field**
definition and use, 13, 78
**PCBB (privileged context block base) register**
address of HWPCB, 21, 75
use by PALcode, 21
**PCC (process cycle counter).** *Consult* Alpha Architecture Reference Manual.
as part of hardware context, 21
**per kernel thread area.** *See* PKTA.
**per-CPU database**
fields used by idle loop, 19–20
pointer to current kernel thread in, 15
scheduling-related fields, 11, 18–20
names and values, 19 (fig.)
**performance monitoring enable register.** *See* PME register.
**PFCDEFAULT parameter (SYSGEN)**
effect on pagelet fault cluster factor for sections, 316
**PFN$W_PT_VAL_CNT field**
L1PT initial value, 286
L1PT value, 287
L2PT initial value, 287
**PFNMAP (map to specific physical pages privilege).** *See* privileges.
**PFW (page fault wait state).** *See also* kernel thread states; page faults; wait states.
AST implications for, 53
context for, 53
transitions
from PFW to COM or COMO, 36
to PFW from CUR, 36
**PGFIPLHI bugcheck.** *See* bugchecks.
**PHD (process header)**
address in
CTL$GL_PHD, 289
KTB, 11
characteristics and use, 6
copying from during process creation, 266 (fig.)
copying to during process creation, 284
expanded for multithreading, 156
floating-point registers saved in, 3
FRED contained in, 11, 156
HWPCB contained in, 21, 284
index array, set to −1 at process deletion, 430
KTB field that contains the address of, 14
layout of FREDs in, 160 (fig.)
local event flags located in, 83
moving process parameters from PQB into, 291 (fig.)
obtaining information from, $GETJPI operations, 234
scheduling-related fields, 17, 18 (fig.)
structure linkage with PSCANCTX and CWPSSRV, 232 (fig.)
variable part, contents, 284
**PHD$L_AUTHPRI field**
process base priority, 292
use by LOGINOUT, 447
**PHD$L_CPULIM field**
scheduling-related PHD field, 17
**PHD$L_CPUTIM field**
quantum expiration use of, 56
scheduling-related PHD field, 17
**PHD$L_EXTDYNWS field**
number of dynamic WSLEs, 373
**PHD$L_FLAGS2 field**
scheduling-related PHD field, 18
**PHD$L_FREDOFF field**
definition and use, 156
offset from PHD to FRED, 288
**PHD$L_FREP1VA field**
first free virtual address in P1 space, 289
**PHD$L_IMGCNT field**
definition and use, 372
tested prior to LIB$SPAWN completion, 465
**PHD$L_LEFC_0 field**
definition and use, 83
**PHD$L_LEFC_1 field**
definition and use, 83
**PHD$L_PSCANCTX_SEQNUM field**
definition and use, 231
**PHD$L_PST_BASE_OFFSET field**
offset to end of PST, 288
**PHD$L_PTCNTMAX field**
number of page table pages with nonzero entries, 288
**PHD$L_PTCNTVAL field**
number of page table pages with valid pages, 286
**PHD$L_TIMREF field**
adjusted by SCH$WAIT_PROC, 49
**PHD$L_WSFLUID field**
initialized during process creation, 290
**PHD$L_WSSIZE field**
initialized during process creation, 290

**PHD$PQ_P1_FIRST_FREE_VA field**
first free P1 space address, 289
**PHD$Q_AUTHPRIV field**
authorized privilege mask, 292
process privilege mask, use and routines that manipulate it, 380(table)
use by LOGINOUT, 447
**PHD$Q_FREE_PTE_COUNT field**
number of available process-private PTES, 289
**PHD$Q_IMAGPRIV field**
process privilege mask, use and routines that manipulate it, 380(table)
**PHD$Q_LEFC field**
definition and use, 83
**PHD$Q_PRIVMSK field**
process privilege mask, 292
use and routines that manipulate it, 380(table)
use by LOGINOUT, 447
**PHD$Q_PSCANCTX_QUEUE field**
definition and use, 231
**PHD$V_NO_WS_CHNG bit (PHD$L_FLAGS field)**
definition and use, 182
**PHDUTL module**. *See* MMG$IMGRESET routine.
**PHV$GL_PIXBAS cell**. *See* PHD.
**physical memory**. *See also* memory.
reducing demand for, by sharing linkage sections, 359
size determined by shell constants and SYSGEN parameters, 285
**PHY_IO (do physical I/O privilege)**. *See* privileges.
**PID (process identifier)**. *See also* EPID; IPID.
characteristics and use, 225
fabricating, 279
index field, characteristics and use, 242
term definition, 5
**piggyback special kernel mode AST**. *See* PKAST.
**pixscan mechanism**
priority boosts handled by, 66–68
**PIXSCAN parameter (SYSGEN)**
pixscan mechanism use of, 66
**PKAST (piggyback special kernel mode AST)**. *See also* AST.
flag that specifies thread-safe, 196
inner mode AST delivery, 199
inner mode semaphore use by, 201
use by lock granting routine when two ASTs must be delivered, 135
**PKTA (per kernel thread area)**
characteristics and field definitions, 161–163
characteristics and use, 7
KTB field that contains address of, 161
layout, 163(fig.)
location, 162
TEB field that contains address of, 213
user-writable fields, 162–163
virtual address space, creating, 171
**PKTA vector**
location, 162
**PKTA$Q_STACK array**
initial values of per kernel thread stack pointers, 162
**PKTA$Q_STACKLIM array**
lowest limit of per kernel thread stacks, 162
**PLV (privileged library vector)**
activating image with, 377
if contained in private section, EXE$IMGACT operations, 337
image activator operations, 337
IMCB field that specifies, 319
privileged shareable image contains, 346
rundown routines listed in, 377
**PME (performance monitoring enable) register**
as part of hardware context, 21
**pool**
nonpaged
allocating, wait state characteristics and use, 39
depletion causes kernel thread resource wait, 37
reclamation. *See* reclamation.
paged
allocating, wait state characteristics and use, 39
**POOLCHECK bugcheck**. *See* bugchecks.
**Portable Operating System Interface**. *See* POSIX.
**POSIX (Portable Operating System Interface)**
incompatible with kernel threads, 164
minimal process, creating, 265
notification of
address space deletion, 425
process rundown, 422
**POSTEF module**. *See* SCH$POSTEF routine.
**PPD$T_FILENAME field**
definition and use, 444
**PQB (process quota block)**
copying during process creation, 266(fig.)
function, 265
layout and field contents, 267(table)
moving process parameters from, 291(fig.)
PCB field that points to, 276
use by EXE$CREPRC, 269
**PQL_D*quota-name* parameter (SYSGEN)**
effect on quota tables, 277
**PQL_DWSDEFAULT parameter (SYSGEN)**
use during working set initialization, 290
**PQL_M*quota-name* parameter (SYSGEN)**
effect on quota tables, 277
**$PRCTERM (Process Termination Notification system service)**
characteristics and use, 291
**preemption of current kernel thread**
control flow, 60–63

**preemption of current kernel thread** *(continued)*
delayed, 256
tests for, 63
**primary page file.** *See* SYS$SPECIFIC:[SYS-EXE]PAGEFILE.SYS.
**primary swap file.** *See* SYS$SPECIFIC:[SYS-EXE]SWAPFILE.SYS.
**priorities.** *See also* IPL; kernel threads—normal; kernel threads—real-time; preemption of current kernel thread; SSETPRI.
altering, 255–256
base
altering, 28
default, 268
initializing, 28
PCB field that contains, 28
resetting by LOGINOUT.EXE, 28
boosts
associated with process creation, 276
associated with process deletion, 421
associated with system events, 63–66
descriptions, 64 (table)
example, 65 (fig.)
pixscan mechanism, 66–68
characteristics and use, 27–32
COM and COMO kernel thread queues, 15
concepts and mechanisms, 27
current
altering, 28, 72
changed by SCH$CHANGE_CUR_PRIORITY, 58
changing for current kernel thread, 57–59
current reset by SCH$QEND, 57
routines that request change in, 57
vary for a normal kernel thread, 31
data structures, scheduler database component, 11
dynamic adjustment of, 31–32
external
representation characteristics, 27, 28
internal
per-CPU database field that contains, 29
representation characteristics, 27
kernel thread
set at kernel thread creation, 171
KTB fields that describe, 28, 269
normal
no boost for priorities 14 and 15, 63
range, 27
per-CPU database field that contains, 19
preemption rules
affected by PRIORITY_OFFSET, 29
in SCH$AQ_PREEMPT_MASK, 28
real-time range, 27
representation in system displays, 28
as significant scheduling characteristic, 9
summary
altered at process deletion, 430
characteristics and use, 29
use in scheduling kernel threads, 178–180
user mode threads, 216
**PRIORITY_OFFSET parameter (SYSGEN)**
kernel thread priority, 29, 31, 59
**privileged context block base register.** *See* PCBB register.
**privileged library vector.** *See* PLV.
**/PRIVILEGED qualifier.** *See also* privileges.
known image installation, 304
**privileged shareable images.** *Consult* OpenVMS Programming Concepts Manual, OpenVMS Linker Utility Manual.
activated, 346–347
characteristics and use, 302
rundown routines implemented as, 346
user-written system services implemented as, 346
**privileges.** *See also* access modes; ACL; protection; SSETPRV; synchronization.
available to processes, 379
characteristics and use, 379–381
checking, 224, 227
image, KFE field that specifies, 320
initialized during process creation, 268
lock information access restricted by, 139
masks
characteristics and use, 380 (table)
process, 379–381
spawned subprocess, inherited from creator, 463, 464
**PRMCEB (create permanent common event flag clusters privilege).** *See also* privileges.
required to
create permanent event flag clusters, 85
delete permanent event flag clusters, 85
use by
SASCEFC, 88
**PRMGBL (create permanent global sections privilege).** *See* privileges.
**PRMMBX (create permanent mailbox privilege).** *See* privileges.
**proactive memory reclamation**
at event flag wait, 93
from periodically waking processes, 54–55
preventing, 182
SYSGEN parameter that enables, 54
testing for during event flag wait, 92
use by EXE$HIBER_INT, 244
**procedure calling mechanism.** *See* call frames; calling standard.
**procedure descriptors**
bound, characteristics and use, 398

location in shareable image that needs relocation, 352
**procedure frames**
DECthreads, characteristics and use, 212
first
set up on supervisor mode stack by CLI, 453
set up on user mode stack by CLI, 457
oldest user mode set up by EXE$PROCSTRT, 295
OpenVMS Alpha compared to VAX, 396
**procedure signature information**
characteristics and use, 397–398
default, 398
generated in response to /TIE qualifier, 398
with linkage pair, 399
**procedure value**
Alpha compared to VAX, 396
used to identify native or translated procedure, 397
**procedures**. *See also* native procedures; thread-safe procedures; translated procedures.
bound, characteristics and use, 398
bridge, characteristics and use, 398
intermediate, for nonminimal jacketing, 400
jacket, characteristics and use, 398
**process affinities**. *See* CPU affinity.
**process bitmap**
limiting repeated deadlock searches for a process, 146
**process computability**
system services affecting, 243–253
**process context**. *See also* execution context; hardware context; software context.
characteristics and use, 2–7
context block. *See* PSCANCTX.
data structures that describe, 3 (fig.)
saving, 3
**process control block**. *See* PCB.
**process creation**. *See also* kernel threads—creating.
components involved in, 433–434
EXE$CREPRC, control flow, 265–277
EXE$PROCSTRT, control flow, 290–296
OpenVMS components that initiate, list, 433
parameters moved during, 266 (fig.)
phases, 263–264, 264 (fig.)
process identification, 279–283
quotas established during, 277–279
swapper's role in, 283–290
SWP$SHELL_INIT, control flow, 285–290
**process cycle counter**. *See* PCC.
**process deletion**. *See also* kernel threads—deleting.
blocking
by IPL 2 execution, 421
by nonzero PCB$L_DPC, 422
PCB$V_NODELET set for, 419
in context of
caller, 417–421
process being deleted, 421–426
system, 428–430
I/O completion required for, 423
open files closed during, 422
PCB$V_DELPEN set for, 420
processes that own subprocesses
example, 431–432, 431 (fig.)
operations, 430–432
required actions, 417
resulting from image exit, 366, 368
RMS operations, 426
single-image process, 438
stages, 417
steps in, 418 (fig.)
**process header**. *See* PHD.
**process identifier**. *See* PID.
**process index**
characteristics and use, 280
maximum, 283
use with multithreaded processes, 281
**process lock queue**
locating locks owned by a process, 122
**process name**
established by LOGINOUT, 447
spawned subprocess, established by DCL, 462
**process page tables**. *See* page tables.
**process pages**. *See also* address space; memory management; pages.
initial, 285
**process priorities**. *See* priorities.
**process quota block**. *See* PQB.
**process residence state**
flag for, 32
**Process Scan system service**. *See* $PROCESS_SCAN.
**process scheduler**. *See* scheduling.
**process sections**. *See also* global sections.
EISD that describes, 330
initialization of EISD and PTEs for, 331 (fig.)
**process states**. *See* kernel thread states; wait states.
**process suspension**. *See* suspension.
**process unique value**
as part of hardware context, 21
**process-permanent files**
equivalence names, 445
**process-private virtual address space**. *See* P0 space; P1 space; process sections.
**processes**. *See also* fork processes; kernel processes; kernel threads; multithreaded processes; PCB; single-threaded processes.
affecting other, 224–229
attributes established at creation reset by LOGINOUT, 446–447
basic concepts, 1–8
checking privileges of, 227

**processes** *(continued)*
classification, 433
by EXESCREPRC, 265
by LOGINOUT, list of types, 444–445
class-scheduled, transforming into normally scheduled, 78
control and communication
overview, 223
system services available for, 224 (table)
control region, retrieving information from, 238
environment contents, 1
EPID, LKB field that contains, 119
execution contexts, 2, 151
hardware context. *See* hardware context.
hibernation
placing into, 244–245
removing from, 245–246
identifying, 224, 279–283
target, 225
information system services, 229–243
interprocess communications
mechanisms, 259–261
IPID, LKB field that contains, 118
joining a resource domain, operations, 114
kernel threads in same, characteristics shared by, 178
life cycle stages, 7–8
limiting deadlock searches for, 146
locating, 226–227
locks owned by, 122
subprocesses owned by, 430
login sequence
batch, 451–452
interactive, 445–451
network, 452
logout, LOGINOUT control flow, 468–470
minimal, created by POSIX, 265
multiple-image
characteristics and use, 466–468
execution, control flow, 467 (fig.)
name
changing, 257
locating target with, 227
shared by kernel threads, 226
network, LOGINOUT image operations, 452
null, characteristics and use, 10
obtaining information from, $GETJPI operations, 234
outswapping, kernel threads enter SUSPO state, 35
page tables. *See* page tables.
PCB field that specifies multithreading, 12
priority. *See* priorities.
privileges available to, 379
quotas
deductible, 277
establishing for new, 277–279
pooled, 277
storage areas for, 278 (table)
remote, servicing requests for, 230 (fig.)
resuming, 252
rundown
distinguished from image rundown, 370
operations, 370–377
security profile components, 1
servicing requests for remote, 228–229
single-image, 434
characteristics and use, 434–438
deleting, 438
execution, control flow, 435 (fig.), 437 (fig.)
states. *See* kernel thread states.
status, 13
flags defined at process creation, 270 (table)
term definition, 1
terminating, 252–253
top-level, creating on another VMScluster node, 265
top-level, term definition, 2
transferring control to with ATTACH command, 465–466
types, 225
**processor status.** *See* PS.
**$PROCESS_AFFINITY (Modify Process Affinity system service)**
explicit affinity, 43
operations, 254
overview, 253
**$PROCESS_CAPABILITIES (Modify Process User Capabilities system service)**
characteristics and use, 42
operations, 255
overview, 254
**PROCESS_ISD_LIST routine (IMGMAPISD module)**
initialization order computation, control flow, 341–343
**$PROCESS_SCAN context block.** *See* PSCANCTX.
**PROCESS_SCAN module.** *See* EXE$PROCESS_SCAN routine; EXE$PSCAN_LOCKCTX routine; EXE$PSCAN_NEXT_PID routine.
**$PROCESS_SCAN system service**
control flow, 233–234
data structures related to, 229–232
$GETJPI use in conjunction with, 229
operations, 229
subroutines in PROCESS_SCAN_CHECK module, 232
subroutines in PROCESS_SCAN_ITMLST module, 232
**PROCSTRT module.** *See* EXE$CATCH_ALL routine; EXE$PROCSTRT routine.

**program**
term definition, 299
**program counter**. *See* PC.
**program sections**. *See* PSECTs.
**PROTECT linker option**
protecting image sections, 302
**protected image**
characteristics and use, 302
**/PROTECTED qualifier**
known image installation, 305
**protection**. *See also* access modes; ACL; locks; privileges.
common event flag clusters, ORB used for, 86
page protection fixups, 357–358, 358 (fig.)
**Proxy Function lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 500
**Proxy Key locks**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 500–501
**PS (processor status)**
as component of hardware context, 3, 20
**$PSCANBUFDEF macro**
$GETJPI buffer header defined by, 232
**PSCANCTX ($PROCESS_SCAN context block)**
definition and use, 231–232
field definitions, 231–232
layout, 231 (fig.)
structure linkage with PHD and CWPSSRV, 232 (fig.)
**$PSCANCTXDEF macro**
context block header defined by, 231
**PSECTs (program sections)**
attributes, 300
characteristics and use, 300
**pseudo devices**. *See* mailboxes; remote terminals; virtual terminals.
**PSWAPM (change process swap mode privilege)**. *See also* privileges.
required to disable swapping, 257
use by
EXE$CREPRC, 268
$SETSWM, 257
**PSXFR (POSIX fork creation wait)**. *See* resource wait—RSN$_PSXFRK.
**PT (page table) space**
characteristics and use, 4
**PTBR (page table base register)**
as component of hardware context, 3, 20
**PTE (page table entry)**. *See also* address space; page tables; pages.
per-CPU database field that contains address of, 19
**PTHREAD$RTL.EXE image**
characteristics and use, 212
shareable image, 212
**PTHREAD_CONFIG logical name**
effect on multithreading, 153

**$QIO (Queue I/O Request system service)**
$QIOW
kernel thread state change actions, 32, 35
**$QIOW**. *See* $QIO—$QIOW.
**quadword index**
term definition, xiv
**quadwords**
how represented in figures, xiv
**quantum**
expiration
operations, 55–57
initialized by EXE$CREATE_KERNEL_THREAD, 172
kernel threads
characteristics and use, 180–181, 209
at inswap, 181
user mode threads
characteristics and use, 209
expiration, 209–210
KTB field that contains, 209
PCB field that contains, 209
**QUANTUM parameter (SYSGEN)**
effect on initial kernel thread quantum, 180
rescheduling use of, 55
use during kernel thread creation, 172
**Queue Definition and Journal Files lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 504–505
**Queue I/O Request system service**. *See* $QIO.
**queue manager**
Forms Cache lock use by, 505
lock use by, characteristics and use of each lock, 501–506
Master Queue lock use by, 506
Queue Definition and Journal Files lock use by, 504–505
requests creation of batch processes, 439, 441
**Queue Manager Alive lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 504
**Queue ORB lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 506
**queues**
lock timeout, deadlock handling use of, 142
resource name. *See also* locks; RSB.
listhead locations in RSB, 112
RSB fields that form, 111
resource wait, lock requests waiting for cluster communication, 130
**Quota Cache Entry lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 484–485

**quotas**. *Consult* OpenVMS System Manager's Manual.
deductible, returned at subprocess deletion, 424
process
on another node, 278
decremented from limit to zero, 278
deductible, 277
establishing for new, 277–279
pooled, 277
returned at subprocess deletion, 430
storage areas for, 278 (table)

**RC instruction**
characteristics and use, 389
**RDAB (resource domain access block)**
characteristics and field definitions, 115
layout, 116 (fig.)
**RDDB (resource domain data block)**
characteristics and field definitions, 115
layout, 116 (fig.)
locating, 115
use by AUDIT_SERVER process, 115
**RDE (region descriptor entry)**
characteristics and use, 289
**RDPB (resource domain pointer block)**
characteristics and use, 115
initialized by EXE$CREPRC, 275
layout, 116 (fig.)
**$READEF (Read Event Flag system service)**
reading event flags, operations, 102
**Read Event Flag system service**. *See* $READEF.
**read-only global pages**. *See* global sections.
**read-write ordering**
Alpha compared to VAX, 388
**Record lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 494–495
**Record Management Services**. *See* RMS.
**recovery unit journaling**. *See* RU journaling.
**region descriptor**
deleting during process rundown, 374
**region descriptor entry**. *See* RDE.
**register frame**. *See* procedures.
**registers**. *See also* AI register; ASTEN register; ASTSR; DATFX register; floating-point registers; PCBB register; PME register; PTBR; scratch integer registers; SP register.
integer, as part of hardware context, 20
memory-mapping, as part of hardware context, 20
translating from VAX to Alpha, 385 (table)
**$REGISTER_THREAD_MANAGER system service**
control flow, 165–167
initializing DECthreads data, 212
**REI module**. *See* EXE$REI_INIT_STACK routine.
**RELEASE$JKT routine (SYSTEM_SERVICE_DISPATCHER module)**
releasing user mode threads, control flow, 194
**relocation**. *See* address relocation.
**relocation records**
characteristics and use, 352–353
executable image linked /SECTION_BINDING contains, 349
layout, 353 (fig.)
**remote devices**. *See* communications; networks; VMScluster systems.
**remote processes**
servicing requests for, 228–229
**request mode**
locks, characteristics and use, 131
**$RESCHED (Reschedule Current Process system service)**
control flow, 256–257
requesting reschedule with, 46
**Reschedule Current Process system service**. *See* $RESCHED.
**rescheduling interrupt (IPL 3)**. *See also* SCH$INTERRUPT routine.
characteristics and operations, 68–78
interrupt service routine
characteristics and operations, 68
control flow, 68–75
requested by, 69 (fig.)
SCH$QEND, 79
SCH$REMOVE_CPU_CAP, 59
**RESHASHTBL parameter (SYSGEN)**
determining number of resource hash table entries, 113
**/RESIDENT qualifier**
known image installation, 305
shareable image installation, 307
**resource availability**. *See* resource wait.
**resource block**. *See* RSB.
**resource domain access block**. *See* RDAB.
**resource domain data block**. *See* RDDB.
**resource domain pointer block**. *See* RDPB.
**resource domains**
access to, controlled by ACL, 107
accessible by processes in prior versions of OpenVMS, 114
active, RDDB describes, 115
data structures, 116 (fig.)
default process, associating with, 115
establishing security profile for, 115
extended, characteristics and use, 114
ID, used to locate RDDB, 115
identified by number, 107
processes joining, operations, 114
resource, term definition, 471
resources, identifier component, 110
RSB field that contains, 110

system, joined at process creation, 294
UIC-based, 114
**resource hash chains**
characteristics and use, 113
resource hash table relations, 114 (fig.)
**resource hash table**
characteristics and use, 113
location and size, 113
resource hash chain relations, 114 (fig.)
structure, 114 (fig.)
**resource master**
characteristics and use, 109
**resource name string**
characteristics, 471
**resource trees**
characteristics and use, 110
maximum depth, 110, 145
remastering, 124
**resource wait**
disabling, 40
effect of $SETRM system service, 40
identifying, 37
miscellaneous wait states, transition descriptions, 38 (table)
mode
enabling and disabling, PCB flag for, 276
enabling and disabling, use of $SETRWM, 257
names and their meaning, list, 38 (table)
process context during, 54
RSN$_ASTWAIT
available after AST delivery in a multithreaded process, 200
declared available at process deletion, 429
$GETJPI use of, 236
PCB flags that indicate kernel threads waiting for, 197
process deletion, 422
uses of, 38
waiting for I/O completion, 423
waiting for subprocess deletion, 422, 431
RSN$_CLUSRV
placing a kernel thread in, 228, 419
use during process creation, 269
RSN$_CLUSTRAN, placing a kernel thread in, 39
RSN$_CPUCAP, placing a kernel thread in, 34, 39, 60, 72
RSN$_INNER_MODE, placing a kernel thread in, 40
RSN$_MAILBOX, placing a kernel thread in, 39
RSN$_MPLEMPTY, placing a kernel thread in, 39
RSN$_MPWBUSY, placing a kernel thread in, 39, 426
RSN$_NPDYNMEM
meaning, 37
placing a kernel thread in, 39
RSN$_PGDYNMEM, placing a kernel thread in, 39
RSN$_PGFILE, placing a kernel thread in, 39
RSN$_PSXFRK, placing a kernel thread in, 40
RSN$_RWCSV, placing a kernel thread in, 39
RSN$_SCS, placing a kernel thread in, 39
states, 38 (table)
use of KTBSL_EFWM to identify, 14, 87
**resources**. *See also* locks; protection; resource wait; synchronization.
access modes, how determined, 110, 111
accessing through common event flags, 104
characteristics
system-owned, meaning, 471
systemwide, meaning, 472
characteristics and use, 110
conversion queue, characteristics and use, 108
deadlock searches, preventing locks from participating in, 142
deallocating RSBs, 472
differentiating, 110
directory node, 123–124
existence on a particular node, determining, 129
identifier components, 110
kernel thread requesting an incompatible lock, 107
locating, 122
master node, 123
name length, 110
names, meaningful with resource domain, 107
OpenVMS Alpha components use of, overview, 471–473
parent, characteristics and use, 473
position in resource tree, 110
queues
listhead locations in RSB, 112
LKB fields that link into RSB queue, 118
removing LKBs from, 138
RSB fields that form, 111
resource domain
identifier component, 110
term definition, 471
resource master operations in granting locks on, 109
root
characteristics and use, 110
distributed directory, 124
linking subresources with, 112 (fig.)
sharing
locks as mechanism for managing, 107
in VMScluster systems, 472
systemwide
distinguished from system-owned locks, 135
privilege required for accessing, 139
term definition, 471

**resources** *(continued)*
value block, location, 112
VMScluster local node use, RSB field that tracks, 111
wait queue, characteristics and use, 108
**$RESUME (Resume Process system service)**
affects all kernel threads in a process, 246
operations, 252
process state change actions, 36
**Resume Process system service.** *See* $RESUME.
**RM$LAST_CHANCE routine ([RMS]RMS0LSTCH module)**
operations, 427–428
**RMS (Record Management Services).** *See also* system services; *Consult* Guide to OpenVMS File Applications.
exit handler declared by EXE$PROCSTRT, 295
File lock, 486–488
lock modes used by, 487
lock use by, characteristics and use of each lock, 485–495
requesting $SYNCH system service, 91
rundown at image exit, 369–370
rundown at process deletion of
global data buffers, 428
local data buffers, 427–428
**RMS exit handler**
characteristics and use, 369–370
purpose, 369
**RMS$RMSRUNDWN routine ([RMS]RMS0-RNDWN module)**
operations, 426
**RMSBUG bugcheck.** *See* bugchecks.
**[RMS]RMS0LSTCH module.** *See* RM$LAST_CHANCE routine.
**[RMS]RMS0RNDWN module.** *See* RMS$RMS-RUNDWN routine.
**RMS_FILEPROT parameter (SYSGEN)**
process default file protection, 447
**root lock**
$ENQ operations, 115
**round-robin scheduling.** *See* scheduling—round-robin.
**RPTEVT macro**
invoking SCH$REPORT_EVENT routine with, 60
**RS instruction**
characteristics and use, 389
**RSB (resource block)**
characteristics and field definitions, 110–113
contains value block, 109
creating for new resource, operations, 130
deallocating, 472
directory entry, characteristics and use, 124
distribution of, in a VMScluster system, 122
field definitions, 110–132
layout, 111 (fig.)
locating, 122
master, contains zero, 123
overview, 108
parent, address as resource identifier component, 110
preallocated, caching, 112
resource domain information stored in, by $ENQ, 115
**RSB$B_CGMODE field**
lock conversion use of, 131
**RSB$B_GGMODE field**
lock conversion use of, 131
**RSB$L_CSID field**
distinguishing LKB type with, 123
**RSDM mutex**
characteristics and use, 140
use by EXE$SET_RESOURCE_DOMAIN, 141
**RSE module.** *See* EXE$CHK_WAIT_BHVR routine; SCH$CHANGE_CUR_PRIORITY routine; SCH$CHSE routine; SCH$CHSEP routine; SCH$PIXSCAN routine; SCH$REPORT_EVENT routine; SCH$WAKE routine.
SCH$QEND. *See* SCH$QEND routine.
**RSN$_x prefix.** *See* resource wait.
**$RSNDEF macro.** *See also* resource wait.
system resource symbolic names defined in, 37
**RU (recovery unit) journaling**
characteristics and use, 426
**rundown**
RMS
image exit, 369–370
process deletion, 426–428
**rundown routines**
access mode called from, 370
activated privileged library dispatch vector fields related to, 378 (fig.)
clearing addresses of, 377
compared to exit handlers, 377
establishing process-permanent, 367
executive mode
called by DELETE, 421
called by EXE$ERNDWN, 371
called by EXIT_CALLBACK, 367
table address in CTL$GL_USRUNDWN_EXEC cell, 292
image-temporary, characteristics and use, 378
implemented as privileged shareable images, 346
inner mode, characteristics and use, 377–379
kernel mode
called by DELETE, 422
called by EXE$KRNDWN, 371
table address in CTL$GL_USRUNDWN cell, 292
part of privileged shareable image, 377
PLV lists, 377

RMS, 422
system services that call, 378
types, 377
**$RUNDWN (Image Rundown system service).** *See also* $ERNDWN; image rundown; $KRNDWN.
operations, 370
**run-time unit journal file**
closing, 369
files connected to, SYS$RMSRUNDWN operations, 369
**RWAST (AST wait).** *See* resource wait—RSN$_ASTWAIT.
**RWCAP (CPU capability).** *See* resource wait—RSN$_CPUCAP.
**RWCLU (cluster transition).** *See* resource wait—RSN$_CLUSTRAN.
**RWCSV (cluster server process).** *See* resource wait—RSN$_CLUSRV.
**RWMBX (mailbox full).** *See* resource wait—RSN$_MAILBOX.
**RWMPB (modified page writer busy).** *See* resource wait—RSN$_MPWBUSY.
**RWMPE (modified page list empty).** *See* resource wait—RSN$_MPLEMPTY.
**RWNPG (nonpaged pool).** *See* resource wait—RSN$_NPDYNMEM.
**RWPAG (paged pool).** *See* resource wait—RSN$_PGDYNMEM.
**RWPFF (page file space).** *See* resource wait—RSN$_PGFILE.
**RWSCS (distributed lock manager wait).** *See* resource wait—RSN$_SCS.

**S0/S1 space**
characteristics and use, 4
**S2 space**
characteristics and use, 4
**SCH$ACQUIRE_AFFINITY routine (CAPABILITY module)**
acquiring implicit affinity with, 45
**SCH$ADD_CPU_CAP routine (CAPABILITY module)**
changing capabilities with, 42
**SCH$AL_CPU_CAP array**
current CPU capabilities, 42, 59
**SCH$AL_CPU_PRIORITY array**
process priority summary, 29, 50, 71
set by
SCH$CHANGE_CUR_PRIORITY, 58
SCH$FIND_NEXT_PROC, 72
**SCH$AQ_COMH array**
array of computable queue listheads, 15
**SCH$AQ_COMOH array**
array of computable and outswapped queue listheads, 15
**SCH$AQ_PREEMPT_MASK array**
priority preemption rules, 28
**SCH$AQ_WQHDR array**
array of wait queue listheads, 16
**SCH$ASTDEL routine (ASTDEL module)**
operations, 200–201
**SCH$ASTDEL_K routine (ASTDEL module)**
operations, 199–200
**SCH$CHANGE_CUR_PRIORITY routine (RSE module)**
control flow, 58–59
routines that call, 57–58
**SCH$CHSE routine (RSE module)**
control flow, 60–63
**SCH$CHSEP routine (RSE module)**
control flow, 60–63
operations, 79
**SCH$CLASS_IDLE routine (SYSSCHED module)**
operations, 80
**SCH$CLASS_PIXSCAN routine (SYSSCHED module)**
operations, 80
**SCH$CLASS_QEND routine (SYSSCHED module)**
control flow, 79
quantum expiration actions, 57
**SCH$CLEAR_CAPABILITY routine (CAPABILITY module)**
changing kernel thread capabilities with, 42
**SCH$C_*mnemonic* (scheduling state symbolic name)**
list of names, their meaning and value, 34 (table)
**SCH$DELETE_CALLBACK routine (SCHEDULER module)**
control flow, 430
operations, 177, 426
single-image process, operations, 438
**SCH$FIND_NEXT_PROC routine (SCHED_ROUTINES module)**
control flow, 71–72
**SCH$GL_AFFINITY_SKIP cell**
default affinity skip, 275
**SCH$GL_CAPABILITY_SEQUENCE cell**
copying into KTB$L_CAPABILITIES_SEQ field, 43
version of current CPU capabilities, 42
**SCH$GL_DEFAULT_CPU_CAP cell**
default CPU capabilities, 42
**SCH$GL_DEFAULT_PROCESS_CAP cell**
copying into KTB$L_PERMANENT_CAPABILITIES field, 42
default kernel thread capabilities, 275
use by
$PROCESS_CAPABILITIES, 255
$SET_IMPLICIT_AFFINITY, 254
**SCH$GL_IDLE_CPUS cell**
cleared by

**SCH$GL_IDLE_CPUS cell** *(continued)*
cleared by *(continued)*
SCH$FIND_NEXT_PROC, 72
SCH$REPORT_EVENT, 62
SCH$STATE_TO_COM, 71
scheduler database component, 11
updated by SCH$IDLE, 77
**SCH$GL_MAXPIX cell**
maximum PCB vector entry, 280
**SCH$GL_PCBVEC cell**
address of PCB vector, 10
**SCH$GL_PIXWIDTH cell**
size of process index field, 283
**SCH$GL_RESMASK cell**
system resources for which kernel threads are waiting, 37
**SCH$GL_TBS cell**
use by CLASS_SCHEDULER image, 79
**SCH$GL_TBSH cell**
indicates if CLASS_SCHEDULER image is loaded, 57, 67, 77
**SCH$GL_TBSO cell**
use by CLASS_SCHEDULER image, 79
**SCH$GL_TBSW cell**
use by CLASS_SCHEDULER image, 79
**SCH$GQ_ACTIVE_PRIORITY cell**
priority summary, 29, 50, 71
use by
SCH$CHANGE_CUR_PRIORITY, 58
SCH$FIND_NEXT_PROC, 72
**SCH$GQ_CEBHD cell**
listhead for systemwide CEB list, 85 (fig.), 85
**SCH$GQ_COMOQS cell**
summary of computable outswapped queues, 15
use by
SCH$PIXSCAN, 67
SCH$REPORT_EVENT, 62
**SCH$GQ_COMQS cell**
summary of computable queues, 15
use by
SCH$FIND_NEXT_PROC, 71, 72
SCH$PIXSCAN, 67
SCH$REPORT_EVENT, 62
SCH$STATE_TO_COM, 71
**SCH$GQ_LEFOWQ cell**
characteristics and use, 98
**SCH$GQ_LEFWQ cell**
characteristics and use, 98
**SCH$GW_CEBCNT cell**
number of CEBs in systemwide list, 85
**SCH$GW_LOCALNODE cell**
use in constructing EPIDs, 283
**SCH$GW_MAXPIXSEQ cell**
maximum sequence number, 282
**SCH$GW_PROCCNT cell**
number of processes, 276
**SCH$GW_PROCLIM cell**
maximum number of processes permitted, 276
**SCH$IDLE routine (SCHED_ROUTINES module).** *See also* idle loop.
idle loop routine, control flow, 77–78
**SCH$INTERRUPT routine (SCHEDULER module)**
control flow, 70–75
executing idle loop, 76
paths to rescheduling, 69 (fig.)
routines that request, 68–70
routines that transfer into, 70
simulating AST delivery interrupt, 74
**SCH$LOCKR routine (MUTEX module)**
saved PC for mutex MWAIT, 54
**SCH$LOCKR_QUAD routine (MUTEX module)**
saved PC for mutex MWAIT, 54
**SCH$LOCKW routine (MUTEX module)**
saved PC for mutex MWAIT, 54
**SCH$LOCKW_QUAD routine (MUTEX module)**
saved PC for mutex MWAIT, 54
**SCH$PAGEFAULT routine (SCHEDULER module)**
waiting a kernel thread, 53
**SCH$PAGEFAULT_UPCALL_JKT routine (SCHEDULER module)**
upcalls, operations, 205
**SCH$PIXSCAN routine (RSE module)**
pixscan mechanism, control flow, 66–68
**SCH$POSTEF routine (POSTEF module)**
multithreaded process, operations, 210–211
setting event flags, control flow, 100–101
**SCH$QAST routine (ASTDEL module)**
multithreaded process, control flow, 197–198
**SCH$QEND routine (RSE module)**
control flow, 56–57
quantum end signaled by invoking, 56
**SCH$RAVAIL routine (MUTEX module)**
declaring resource available, 38
**SCH$REMOVE_CPU_CAP routine (CAPABILITY module)**
changing capabilities with, 42
control flow, 59
**SCH$REPORT_EVENT routine (RSE module)**
control flow, 60–63
reporting set-priority system event, control flow, 256
setting event flags, operations, 101–102
**SCH$RESOURCE_WAIT routine (MUTEX module)**
saved PC for resource MWAIT, 54
**SCH$RESOURCE_WAIT_PS routine (MUTEX module)**
saved PS for resource MWAIT, 54
**SCH$RESOURCE_WAIT_SETUP routine (MUTEX module)**
waiting a kernel thread, 50

**SCH$SET_AFFINITY routine (CAPABILITY module)**
operations, 44
**SCH$SET_CAPABILITY routine (CAPABILITY module)**
changing kernel thread capabilities with, 42
**SCH$STATE_TO_COM routine (SCHED_ROUTINES module)**
updating scheduler database, control flow, 71
**SCH$THREADQ_END routine (TIMESCHDL module)**
user mode thread quantum end, control flow, 209
**SCH$WAIT_ANY_MODE routine (SCHEDULER module)**
control flow, 49–50
initiating context swap, 50
routines that call, 51–52
**SCH$WAIT_KERNEL_MODE routine (SCHEDULER module)**
initiating context swap, 51
routines that call, 51–52
**SCH$WAIT_KERNEL_MODE_PS routine (SCHEDULER module)**
initiating context swap, 51
routines that call, 51–52
**SCH$WAIT_PROC routine (SCHED_ROUTINES module)**
control flow, 47–49
entered in kernel thread context, 47
event flag wait, operations, 93
waiting a kernel thread, 50
**SCH$WAKE routine (RSE module)**
operations, 245–246
upcalls, operations, 206
**$SCHDWK (Schedule Wakeup system service)**
kernel thread state change actions, 35
operations, 245
**$SCHED (Affect Process Scheduling system service)**
operations, 78
**SCHED spinlock**
held during
system event reporting, 60
waiting a kernel thread, 47
synchronizing
access to scheduler database, 11, 83
use by
DELETE, 426
DELETE_IN_SYS_CONTEXT, 428
EXE$CREATE_KERNEL_THREAD, 172
EXE$CREPRC, 274, 276
EXE$DACEFC, 89
EXE$DELETE_KERNEL_THREAD, 175
EXE$DELPRC, 419
EXE$DEREGISTER_THREAD_MANAGER, 168
EXE$END_MULTITHREAD, 167
EXE$HIBER_INT, 244
EXE$KRNDWN, 376
EXE$KTHREAD_DELETE_CALLBACK, 177
EXE$KTHREAD_DELETE_SELF, 176
EXE$NAM_TO_PCB, 227, 228, 419
EXE$RESCHED, 256
EXE$RESUME, 252
EXE$SINGLE_THREAD, 202, 203
EXE$SOFT_SUSPND_INT, 251
EXE$SS_RELEASE_THREAD, 194
EXE$SS_WAIT_THREAD, 192
EXE$SUSPND, 248
EXE$SYNCH_INT, 97
EXE$WAIT, 93
HARD_SUSPND_AST, 250
SCH$CHANGE_CUR_PRIORITY, 58
SCH$CLASS_IDLE, 80
SCH$FIND_NEXT_PROC, 72
SCH$IDLE, 77
SCH$INTERRUPT, 71
SCH$PIXSCAN, 66
SCH$POSTEF, 100
SCH$QAST, 197
SCH$QEND, 56
SCH$STATE_TO_COM, 71
SCH$WAIT_ANY_MODE, 49
SCH$WAIT_PROC, 47
SUSPND_INTERIM_AST, 249
SWP$SHELL_INIT, 286
system service waiting a kernel thread, 49
**Schedule Wakeup system service.** *See* $SCHDWK.
**scheduler database**
components, 10–20
procedures that update while waiting a kernel thread, 50
synchronizing access to with SCHED spinlock, 11
**SCHEDULER module.** *See* SCH$DELETE_CALLBACK routine; SCH$PAGEFAULT_UPCALL_JKT routine; SCH$WAIT_ANY_MODE routine; SCH$WAIT_KERNEL_MODE routine; SCH$WAIT_KERNEL_MODE_PS routine.
**SCH$INTERRUPT.** *See* SCH$INTERRUPT routine.
**SCH$PAGEFAULT.** *See* SCH$PAGEFAULT routine.
**scheduling.** *See also* CPU affinity; CPU capabilities; kernel thread states; priorities; rescheduling interrupt; scheduler database; swapper; synchronization; wait states.
classes, characteristics and use, 78
dynamics, characteristics and operations, 45–68

**scheduling** *(continued)*
event reporting
during kernel thread state change, control flow, 60–63
paths leading to, 61 (fig.)
SCH$REPORT_EVENT, control flow, 60–63
events that require, 46
FRED fields related to, 17–18
kernel thread characteristics most important to, 9
KTB fields related to, 13–15
overview, 9
PCB fields related to, 11–13
per-CPU database fields related to, 18–20
PHD fields related to, 17
round-robin
operations, 55
when used, 31
states. *See* kernel thread states.
wait queues, header array, 17 (fig.)
**SCHED_ROUTINES module**. *See* SCH$FIND_NEXT_PROC routine; SCH$IDLE routine; SCH$STATE_TO_COM routine; SCH$WAIT_PROC routine.
**scratch integer registers**
characteristics and use, 23
**SCS spinlock**
access to lock database synchronized by, 127
held during dispatch to system-owned locks blocking AST routine, 136
use by
EXE$DEQ, 138
**SCSSYSTEMID parameter (SYSGEN)**
system ID lock naming, 473
**SDA (System Dump Analyzer)**. *Consult* OpenVMS Alpha System Dump Analyzer Utility Manual.
image mapping, example, 308–311
mapping executable image with shareable images, 309 (fig.)
**section descriptors**. *See also* EISD.
characteristics and use, 324
**sections**
virtual address space managed by, 302–303
**/SECTION_BINDING qualifier**
executable images, 349
known image installation, 305, 306
shareable image installation, 307
**SECURITY_AUDITNG module**. *See* NSA$DE-ACCESS_AUDIT routine.
**semaphores**. *See also* inner mode semaphore; mutexes.
common event flags as, 104
**Send to Job Controller system service**. *See* $SNDJBC.
**sequence vector**
characteristics and use, 280–282
sized by MAXPROCESSCNT, 280
updated at process deletion, 428
**Set AST Enable system service**. *See* $SETAST.
**Set Event Flag system service**. *See* $SETEF.
**Set Exception Vector system service**. *See* $SETEXV.
**Set Priority system service**. *See* $SETPRI.
**Set Privileges system service**. *See* $SETPRV.
**Set Process Name system service**. *See* $SETPRN.
**SET PROCESS/PRIORITY command**
kernel thread priority changed by, 28
**Set Resource Domain system service**. *See* $SET_RESOURCE_DOMAIN.
**Set Resource Wait Mode system service**. *See* $SETRWM.
**SET SECURITY command**
establishing security profile for resource domain, 115
**Set Swap Mode system service**. *See* $SETSWM.
**Set System Service Failure Exception Mode system service**. *See* $SETSFM.
**Set System Service Filter system service**. *See* $SETSSF.
**Set Time lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 474
**Set Time system service**. *See* $SETIME.
**SET WORKING_SET command**. *See* working set.
**$SETAST (Set AST Enable system service)**. *See* AST.
**$SETEF (Set Event Flag system service)**. *See also* event flags.
setting event flags, operations, 100
**$SETEXV (Set Exception Vector system service)**. *See* condition handlers.
**$SETIME (Set Time system service)**
capabilities, use of, 44
**$SETIMR (Set Timer system service)**. *See* timers—requests.
**SETIPL macro**. *See* IPL.
**$SETPRI (Set Priority system service)**. *See also* priorities.
kernel thread priority changed by, 28
operations, 255–256
**$SETPRN (Set Process Name system service)**
operations, 257
**SETPRV (set any privilege bit privilege)**. *See also* privileges.
use by
EXE$CREPRC, 268
$SETPRV, 381
**$SETPRV (Set Privileges system service)**. *See also* privileges.
operations, 381
use when spawning subprocess, 463

**$SETRWM (Set Resource Wait Mode system service)**
disabling resource waits, 40
operations, 257
**$SETSFM (Set System Service Failure Exception Mode system service)**
operations, 257
**$SETSWM (Set Swap Mode system service)**
operations, 257
**$SET_IMPLICIT_AFFINITY (Modify Process Implicit Affinity system service)**
characteristics and use, 45
operations, 254
overview, 253
**$SET_RESOURCE_DOMAIN (Set Resource Domain system service)**
operations, 114, 140–141
**$SEVERITY symbol**
final image severity, 458
**SFSB (shared file synchronization block)**
characteristics and field definitions, 487–488
**SGN$GW_PIXSCAN cell.** *See* PIXSCAN (SYSGEN parameter).
**SHARE (assign channels to nonshared devices privilege).** *See* privileges.
**/SHARE qualifier**
known image installation, 305
**shareable address data**
images installed with, 306, 324, 358
activating, 346
address relocation and fixup processing, 350
flag that indicates, 324
images installed without, 324
**shareable image installation**
/RESIDENT qualifier, 307
/SECTION_BINDING qualifier, 307
/SHARED qualifier, 306
**shareable image list entry.** *See* SHL.
**shareable images.** *See also* images.
assigning address space, 358
building, 302–303
image mapping, example, 308–311
image activator, control flow, 335–338
IMCB field that contains old-format name of global sections of, 318
initializing, computing proper order of, 340–343
installed
/RESIDENT, activating, 307
/SHARE=ADDRESS_DATA/RESIDENT, activating, 307, 346, 358
/SHARED, virtual address space, 306–307
list, 351–352, 352 (fig.)
term definition, 301
**shareable linkage**
term definition, 359
**shared file synchronization block.** *See* SFSB.
**/SHARED qualifier**
executable image installation, 306
shareable image installation, 306
**SHELLxK module.** *See also* processes; SWP$SHELL_INIT routine.
constants, initial process size determined by, 285
defining kernel thread private area, 162
layout, characteristics and use, 285
**shelving bits**
characteristics and use, 271
**SHL (shareable image list entry)**
characteristics and use, 351–352
**SHUTDOWN.COM command file.** *Consult* OpenVMS System Manager's Manual.
**single threading**
terminating, by EXE$SINGLE_THREAD, 203
**single-image processes.** *See* processes—single-image.
**single-threaded processes**
EXE$EXIT operations, 366
quantum effect on, 55
synchronization assumptions, 184
wait services actions, 92
**$SINGLE_THREAD system service**
arguments, 202
control flow, 202–203
CTRL/Y processing, operations, 459
operations, 201
**$SINGLE_THREAD_INT system service**
operations, 203
**SMGSHR.EXE image**
image mapping, example, 309–310
**SMISERVER Main lock.** *See also* locks used by OpenVMS Alpha components.
characteristics and use, 507
**SMP (symmetric multiprocessing) systems.** *See also* memory management; synchronization.
multiple kernel threads of a process, 2
priority data structures, 29, 30 (fig.)
scheduling
considerations, 29
constraints, 9
supported by per-CPU database, 11
**$SNDJBC (Send to Job Controller system service).** *See also* batch processes; job controller.
$SNDJBCW
use by LOGINOUT, 441, 451
**$SNDJBCW (Send to Job Controller and Wait system service).** *See* $SNDJBC.
**soft suspension.** *See* suspension.
**software conditions.** *See* condition handlers.
**software context**
characteristics and use, 5–7
**software interrupts.** *See* AST delivery interrupts; fork interrupts; rescheduling interrupt; software timers.

**software IPLs**. *See* IPL.
**software timers**
interrupt (IPL 7)
role in quantum expiration, 209
**software-vectored condition handlers**. *See* condition handlers.
**SOFT_SUSPND_AST routine (SYSPCNTRL module)**
supervisor mode AST procedure, operations, 251
**$SOFT_SUSPND_INT system service**
operations, 251–252
**source modules**
facilities contain, xii
names, xii
**SP (stack pointer) register**
as part of hardware context, 20
**SPAWN command**. *See also* ATTACH command; processes.
operations, 461–465
**special conditions**. *See* condition handlers.
**special kernel mode AST**
$GETJPI use of, 241–242
obtaining information about an outswapped process with, 238
queuing, after granting a lock, reasons for, 135
**spinlocks**. *See also* IPL; locks; synchronization.
MMG. *See* MMG spinlock.
releasing
periodically to prevent timeouts, 426
SCHED. *See* SCHED spinlock.
SCS. *See* SCS spinlock.
**SPLACQERR bugcheck**. *See* bugchecks.
**SPLIPLHIGH bugcheck**. *See* bugchecks.
**SPLIPLLOW bugcheck**. *See* bugchecks.
**SPLRELERR bugcheck**. *See* bugchecks.
**SPLRSTERR bugcheck**. *See* bugchecks.
**SPTE (system page table entry)**. *See also* PTE.
allocated for lock IDs, 121
**SS$_ACCONFLICT error status**
returned by EXE$IMGACT, 336
**SS$_ACCVIO error status**
returned by
EXE$ASCEFC, 88
EXE$CREATE_KERNEL_THREAD, 170
EXE$CREPRC, 265, 269
EXE$DLCEFC, 90
EXE$ENQ, 127
EXE$INIT_MULTITHREAD, 164
EXE$PROCESS_SCAN, 233
EXE$REGISTER_THREAD_MANAGER, 166
EXE$SET_RESOURCE_DOMAIN, 140
EXE$SINGLE_THREAD, 202
EXE$SYNCH_INT, 97
**SS$_BADIMGHDR error status**
returned by EXE$IMGACT, 329, 336
**SS$_BADPARAM error status**
returned by
SCMEXEC, 401
SCMKRNL, 401
EXE$CREATE_KERNEL_THREAD, 171
EXE$ENQ, 136
EXE$INIT_MULTITHREAD, 164
EXE$REGISTER_THREAD_MANAGER, 166
EXE$SET_RESOURCE_DOMAIN, 140
TIE$CMEXEC, 412
TIE$CMKRNL, 412
**SS$_BUGCHECK error status**. *See* bugchecks.
**SS$_DEADLOCK error status**
returned by EXE$ENQ, 141
**SS$_DEBUG signal**
generated by CLI in response to DEBUG command, 364, 460
**SS$_DUPLNAM error status**
returned by EXE$CREPRC, 275
**SS$_EXASTLM error status**
returned by EXE$GETJPI, 237
**SS$_EXDEPTH error status**
returned by EXE$ENQ, 130
**SS$_EXENQLM error status**
returned by EXE$ENQ, 129
**SS$_EXPRCLM error status**
returned by EXE$CREPRC, 274
**SS$_EXQUOTA error status**
returned by
EXE$ASCEFC, 88
EXE$CREPRC, 272, 277
**SS$_ILLEFC error status**
returned by
EXE$ASCEFC, 88
EXE$DACEFC, 89
EXE$WAIT, 92
SCH$POSTEF, 100
**SS$_ILLRSDM error status**
returned by EXE$ENQ, 128, 129
**SS$_INVSRQ error status**
returned by EXE$PROCESS_SCAN, 233
**SS$_IVBUFLEN error status**
returned by EXE$ENQ, 127
**SS$_IVLOCKID error status**
returned by
EXE$DEQ, 121, 138
EXE$ENQ, 121, 128
**SS$_IVLOGNAM error status**
returned by
EXE$ASCEFC, 88
EXE$CREPRC, 269
EXE$NAM_TO_PCB, 227
**SS$_IVSSRQ error status**
returned by
$SETEXV, 401
TIE$DCLCMH, 412

**SS$_IVSTSFLG error status**
returned by EXE$CREPRC, 268
**SS$_MAXKTHREADS error status**
returned by EXE$CREATE_KERNEL_THREAD, 170
**SS$_MTHREADACTIVE error status**
returned by
EXE$DEREGISTER_THREAD_MANAGER, 168
EXE$INIT_MULTITHREAD, 164
EXE$REGISTER_THREAD_MANAGER, 165
**SS$_NOCALLTRANS error status**
returned by EXE$_NATIVE_TO_TRANSLATED, 402
**SS$_NOCALLTRANS_SUP exception**
signaled by EXE$NATIVE_TO_TRANSLATED, 402
**SS$_NODELETE error status**
returned by
EXE$DELETE_KERNEL_THREAD, 175
EXE$DELPRC, 419
**SS$_NODOMAIN error status**
returned by EXE$ENQ, 128
**SS$_NOMOREPROC error status**
returned by
EXE$GETJPI, 235, 242
EXE$PSCAN_NEXT_PID, 237
**SS$_NOMORETHREAD error status**
returned by EXE$GETJPI, 235, 243
**SS$_NOMULTITHREAD error status**
returned by
EXE$CREATE_KERNEL_THREAD, 170
EXE$DELETE_KERNEL_THREAD, 175
EXE$DEREGISTER_THREAD_MANAGER, 168
EXE$END_MULTITHREAD, 167
EXE$INIT_MULTITHREAD, 164
EXE$REGISTER_THREAD_MANAGER, 165
**SS$_NONEXPR error status**
returned by
EXE$GETJPI, 239
EXE$NAM_TO_PCB, 227
EXE$SUSPND, 248
SCH$POSTEF, 100
**SS$_NOPRIV error status**
returned by
EXE$ASCEFC, 88
EXE$CREPRC, 268
EXE$DLCEFC, 90
EXE$NAM_TO_PCB, 228
EXE$SET_RESOURCE_DOMAIN, 140
EXE$SUSPND, 247
**SS$_NOSHRIMG error status**
returned by IMG$PRVSHRIMG, 347
**SS$_NOSLOT error status**
returned by EXE$CREPRC, 276
**SS$_NOSUCHNODE error status**
returned by
EXE$CREPRC, 269
EXE$NAM_TO_PCB, 227
**SS$_NOSUSPEND error status**
returned by EXE$SUSPND, 248
**SS$_NOSYSLCK error status**
returned by EXE$ENQ, 129
**SS$_NOTALLPRIV error status**
returned by EXE$SETPRIV, 381
**SS$_NOTHREADMAN error status**
returned by EXE$DEREGISTER_THREAD_MANAGER, 168
**SS$_NOTINSTALL error status**
returned by EXE$IMGACT, 332
**SS$_NOTQUEUED error status**
returned
by EXE$ENQ, 130, 133, 136
on illegal lock request, 125, 126
**SS$_PARNOTGRANT error status**
returned by EXE$ENQ, 128
**SS$_PRIVINSTALL error status**
returned by EXE$IMGACT, 328, 335, 336
**SS$_PROTINSTALL error status**
returned by EXE$IMGACT, 337
**SS$_REMOTE_PROC error status**
returned by
CWPS routine, 228
EXE$NAM_TO_PCB, 227, 237, 247, 253, 255, 419
EXE$PSCAN_NEXT_PID, 237
**SS$_RSDMNOTFOU error status**
returned by EXE$SET_RESOURCE_DOMAIN, 140
**SS$_SHRIDMISMAT error status**
returned by EXE$IMGACT, 336
**SS$_SUBLOCKS error status**
returned by EXE$DEQ, 138
**SS$_SUSPENDED error status**
returned by $GETJPI, 239
**SS$_SYNCH error status**
returned by EXE$ENQ, 130
**SS$_SYNCSTS error status**
returned by lock management system services, 127
**SS$_SYSVERDIF error status**
returned by EXE$IMGACT, 329
**SS$_TRANSCALLER error status**
returned by EXE$TRANSLATED_TO_NATIVE, 403
**SS$_UNASEFC error status**
returned by
EXE$WAIT, 92
SCH$POSTEF, 101
**SS$_UNSUPPORTED error status**
returned by

**SS$_UNSUPPORTED error status** *(continued)*
returned by *(continued)*
EXE$ENQ, 127
EXE$INIT_MULTITHREAD, 164
EXE$SET_RESOURCE_DOMAIN, 140
**SS$_WAIT_CALLERS_MODE status**
returned by
EXE$END_MULTITHREAD, 167
EXE$SOFT_SUSPND_INT, 251
EXE$SS_WAIT_THREAD, 192
EXE$SYNCH_INT, 97
EXE$WAIT, 93
SCH$WAIT_PROC, 245
system service waiting a kernel thread, 47, 49, 51
**SS$_WASCLR status**
returned by
EXE$CLREF, 102
EXE$READEF, 102
SCH$POSTEF, 101
**SS$_WASSET status**
returned by
EXE$CLREF, 102
EXE$READEF, 102
EXE$SYNCH_INT, 97
SCH$POSTEF, 101
system services when EFN$C_ENF is passed, 83
wait system services, 91, 92
**SSRVEXCEPT bugcheck**. *See* bugchecks.
**SSVECFULL bugcheck**. *See* bugchecks.
**$SS_RELEASE_THREAD system service**
control flow, 194–195
**$SS_WAIT_THREAD system service**
control flow, 192
**stack**. *See also* canonical kernel stack; executive stack; kernel stack; procedures; supervisor stack; user stack.
determining boundaries of, 162
kernel thread, one per access mode, 3
kernel, contents at process creation, 284 (fig.)
**stack frames**. *See* procedures.
**STACK linker option**
specifying size of user stack, 339
**stack pointer register**. *See* SP register.
**$STATEDEF macro**
symbolic scheduling names defined in, 32
**STATENTSVD bugcheck**. *See* bugchecks.
**states**. *See* kernel thread states.
**status block**
recommended usage, 102
synchronous system services return path handling of, 94–97
use with
asynchronous operations, 102
synchronous operations, 103
**status codes**. *See* SS$_x and other specific status names.
**$STATUS symbol**
final image status, 458
**STOP command**
contrasted to image exit, 461
operations, 461
**store-conditional instruction**. *See* load-locked/store-conditional mechanism.
**STx_C (store-conditional) instruction**. *See* load-locked/store-conditional mechanism.
**sublocks**. *See also* locks.
characteristics and use, 119 (fig.)
dequeuing, 137
**SUBMIT command**
creating batch processes
arguments passed to SCREPRC, 441 (table)
operations, 441, 442 (fig.)
job controller process creation of batch processes in response to, 439
**subprocesses**
deleting, 422
returning deductible quotas to owner process, 424
deleting a process that owns
example, 431–432, 431 (fig.)
operations, 430–432
interactive, DCL commands that create and connect with, 461
locating, 430
spawning with SPAWN command, 461–465
**subresources**. *See also* resources.
characteristics and use, 110
linking root resources with, 112 (fig.)
**supervisor mode**. *See also* access modes.
CLI execution in, 453
exit handler, CLI declaration of, 453
stack pointer, as part of hardware context, 20
suspension
AST queuing during, 249
characteristics, 246, 248
SUSPND_INTERIM_AST actions, 249
**supervisor mode stack**. *See* supervisor stack.
**supervisor stack**
pages, protection, 161
**SUSP (suspended wait state)**. *See also* kernel thread states; suspension; wait states.
context for, 53–54
placing a kernel thread in, 246, 251
quiescent kernel threads wait in, 201
transitions
from SUSP to COM, 36, 420
to SUSP from other states, 35
**suspended wait state**. *See* SUSP.

**suspension**. *See also* hibernation.
access mode used to categorize SUSP and SUSPO states, 35
AST queuing during, 247
constraints, 246
hard
at process deletion, 420
term definition, 246
hibernation compared to, 247
inner mode semaphore use during, 247
kernel mode
AST queuing during, 249
characteristics, 246, 248
SUSPND_INTERIM_AST actions, 249
soft
at process deletion, 420
term definition, 246
supervisor mode
AST queuing during, 249
characteristics, 246, 248
SUSPND_INTERIM_AST actions, 249
**$SUSPND (Suspend Process system service)**
affects all kernel threads in a process, 180, 246
control flow, 247–249
kernel thread state change actions, 32, 35
**SUSPND_INTERIM_AST routine (SYSPCNTRL module)**
kernel mode AST procedure, operations, 249–250
**SUSPO (suspended outswapped wait state)**. *See also* kernel thread states; SUSP; suspension; wait states.
outswapping causes kernel threads to enter, 35
transitions
to SUSPO from other states, 35
from SUSPO to COMO, 36
**SWAPFILE.SYS**. *See* SYS$SPECIFIC:[SYSEXE]SWAPFILE.SYS.
**swapper**. *See also* inswap; memory management; modified page writer; outswap; page faults; scheduling; swapper trimming; SWP$SHELL_INIT routine; wait states.
awakened by
DELETE_IN_SYS_CONTEXT, 430
SCH$CHSEP, 62
SCH$QEND, 57
contains no control region or JIB, 238
PCB, location of, 10
preparing for process inswap, control flow, 285
process creation
operations, overview, 263
role in, 283–290
trimming. *See* swapper trimming.
**swapper trimming**
multithreaded processes, 181
preventing, 182
**swapping**. *See* inswap; outswap; swapper.
**SWP$GL_MAP cell**
starting address of swapper map, 285
**SWP$GL_SHELLSIZ cell**
number of pages to be allocated to process, 285
**SWP$SHELL_INIT routine (SHELLxK module)**
control flow, 285–290
initializing
L1PT, control flow, 286
L2PT, control flow, 287
L3PT, control flow, 287
process creation, operations, 283–284
**SWPCTX PALcode routine**
conditions upon entering, 75
control flow, 75–76
operations, 22–23
routines that use, 76
**symbiont processes**. *See* job controller.
**symbol vector**
EIHD fields that contain information about, 313
image locations requiring relocation contained in, 352
IMCB fields that contain information about, 319
term definition, 302
**symbols**. *See* DCL symbols; global symbols; map files; naming conventions; relocatable symbols; universal symbols.
**symmetric multiprocessing**. *See* SMP systems.
**$SYNCH (Synchronize system service)**
control flow, 94–96
handling kernel threads that use EFN$C_ENF, 84
kernel thread state change actions, 32, 35
program flow example, 98, 99 (fig.)
requesting event flag wait, 91
$SYNCH_INT
internal kernel mode form of service, control flow, 97
testing for event completion, 82
**synchronization**. *See also* event flags; fork processes; load-locked/store-conditional mechanism; locks; mutexes; SMP systems; spinlocks.
CEB list access, with EXE$GL_CEBMTX mutex, 85
event flags, characteristics and use, 81–82
interprocess, through common event flags, 104
multithreaded mechanisms, 184
multithreaded process, 183–204
process use of common event flags for, 104
scheduler database access
SCHED spinlock use for, 11
single-threaded process, 184
VMScluster resources, lock management as fundamental technique, 109
**Synchronize system service**. *See* $SYNCH.
**$SYNCH_INT**. *See* $SYNCH.

**SYS$DISK logical name**
translated at process creation, 268
**SYS$GL_BJOBCNT cell**
batch job count, 276
**SYS$GL_IJOBCNT cell**
interactive job count, 276
**SYS$INPUT logical name**
CLI processing use of, 453
**SYS$LOGIN logical name**
use by LOGINOUT, 448
**SYS$NODE logical name**
use by LOGINOUT, 449
**SYS$RMSRUNDWN system service**
classes of files handled by, 369
control flow, 369–370
operations, 369
**SYS$SPECIFIC:[SYSEXE]PAGEFILE.SYS (primary page file).** *See* memory management; page faults.
**SYS$SPECIFIC:[SYSEXE]SWAPFILE.SYS (primary swap file).** *See* inswap; memory management; outswap; swapper.
**SYS$SYSTEM:OPCCRASH.EXE.** *See* [OPCOM]OPCCRASH module.
**SYS$WELCOME logical name**
use by LOGINOUT, 449
**SYSASCEFC module.** *See* EXE$ASCEFC routine.
**SYSASTCON module.** *See* EXE$SETAST routine.
**SYSCREDEL module**
MMG$CRETVA_K. *See* MMG$CRETVA_K routine.
**SYSCREPRC module.** *See* EXE$CREPRC routine.
**SYSDELPRC module.** *See* DELETE_IN_SYS_CONTEXT routine; EXE$DELPRC routine; EXEC_RUNDOWN_AST routine.
**SYSENQDEQ module.** *See* EXE$DEQ routine; EXE$ENQ routine; LCK$GRANT_LOCK routine.
**SYSEVTSRV module.** *See* EXE$CLREF routine; EXE$READEF routine.
**SYSEXIT module.** *See* EXE$EXIT routine; EXE$EXIT_INT routine; EXIT_CALLBACK routine.
**SYSFASTIO module.** *See* EXE$FANDLE_RUNDOWN routine.
**SYSFORCEX module.** *See* EXE$FORCEX routine.
**SYSGBL (create systemwide global sections privilege).** *See* privileges.
**SYSGEN (System Generation utility)**
*Consult* OpenVMS System Management Utilities Reference Manual, 519
**SYSGEN parameters**
initial process size determined by, 285
term definition, xiii
use when creating process on another node, 279
**SYSGETJPI module**
EXE$GETJPI. *See* EXE$GETJPI routine.
**SYSGETLKI module.** *See* EXE$GETLKI routine.
**SYSIMGACT module.** *See* EXE$IMGACT routine.
**SYSIMGFIX module.** *See* EXE$IMGFIX routine; IMG$PRVSHRIMG routine.
**SYSIMGSTA module.** *See* EXE$IMGSTA routine.
**SYSJACKET module.** *See* EXE$NATIVE_TO_TRANSLATED routine; EXE$TRANSLATED_TO_NATIVE routine.
**SYSLCK (lock systemwide resources privilege).** *See also* privileges.
accessing system name space, 114
lock information access permitted by, 139
locking a resource, 113
use by EXE$ENQ, 129, 472
use when creating a system lock, 115
**[SYSLOA]CWPS_SERVICE_RECV module.** *See* CWPS$SRCV_GETJPI_AST routine.
**SYSMAN (System Management utility)**
lock use by, characteristics and use of each lock, 507
**SYSNAM (insert in system logical name table privilege).** *See* privileges.
**SYSPARPRC module.** *See* EXE$RESCHED routine.
**SYSPCNTRL module.** *See* EXE$CHECKS_PROCESS_ACCESS routine; EXE$CVT_EPID_TO_IPID routine; EXE$CVT_EPID_TO_KTB routine; EXE$CVT_EPID_TO_PCB routine; EXE$CVT_IPID_TO_EPID routine; EXE$CVT_IPID_TO_KTB routine; EXE$CVT_IPID_TO_PCB routine; EXE$HIBER routine; EXE$HIBER_INT routine; EXE$NAM_TO_PCB routine; EXE$RESUME routine; EXE$SETPRN routine; EXE$SOFT_SUSPND_INT routine; EXE$SUSPND routine; EXE$WAKE routine; SOFT_SUSPND_AST routine; SUSPND_INTERIM_AST routine.
**SYSPRV (access objects via system protection privilege).** *See* privileges.
**SYSRUNDWN module.** *See* EXE$ERNDWN routine; EXE$KRNDWN routine; EXE$RUNDWN routine.
**SYSSCHED module.** *See* SCH$CLASS_IDLE routine; SCH$CLASS_PIXSCAN routine; SCH$CLASS_QEND routine.
**SYSSCHEVT module**
EXE$SCHDWK. *See* EXE$SCHDWK routine.
**SYSSETMOD module**
system service routines for changing process characteristics, 257
**SYSSETPRI module.** *See* EXE$SETPRI routine.
**SYSSETPRV module.** *See* EXE$SETPRV routine.

**SYSSYNCH module**. *See* EXE$SYNCH routine; EXE$SYNCH_INT routine.
EXE$SYNCH_LOOP. *See* EXE$SYNCH_LOOP routine.
**system context**
entered for process deletion, 426
kernel thread deletion in, 177
location of HWPCB, 20
process deletion in, 428–430
**system crashes**. *See* bugchecks.
**system domain**
process associated with, 115
**System Dump Analyzer**. *See* SDA.
**System ID lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 473–474
**system initialization**. *See also* SYSGEN parameters.
image activation at, 344
**system page table entry**. *See* SPTE.
**system pages**. *See* address space; pages; PTE; SPTE; system space.
**system resources**. *See* resources.
**system service intercept vector**
image activator operations, 337
**system services**. *See also* RMS.
for automatic jacketing, 401–404
EFN$C_ENF use by, 83–84
images, activating and exiting, 299
lock management system, operations, 108, 126–140
process computability, 243–253
process information, 229–243
requesting from translated images, 411–413
requiring explicit intervention when requested from translated image, 411
setting process software status, 257
that affect kernel thread affinity, 254
that affect kernel thread capabilities, 255
that call rundown routines, 378
that initiate and terminate multithreading, 163–164
that use event flags, 82
that use locks, 108
user-written, implemented as privileged shareable images, 346
**SYSTEM_CHECK parameter (SYSGEN)**
recording inner mode semaphore information, 187
**SYSTEM_SERVICE_DISPATCHER module**. *See* IMSEM_ACQUIRE routine; IMSEM_RELEASE routine; RELEASE$JKT routine; WAIT$JKT routine.
EXE$CMODEXEC. *See* EXE$CMODEXEC routine.
EXESCMODKRNL. *See* EXE$CMODKRNL routine.
**SYSTEM_SERVICE_IMSEM module**. *See* EXE$SS_RELEASE_THREAD routine; EXE$SS_WAIT_THREAD routine.
**SYSWAIT module**. *See* EXE$WAIT routine; EXE$WAITFR routine; EXE$WFLAND routine; EXE$WFLAND_COMMON routine; EXE$WFLAND_INT routine; EXE$WFLOR routine; EXE$WFLOR_COMMON routine; EXE$WFLOR_INT routine.
**[SYS] facility**
base image source modules contained in, xii
many executive image source modules contained in, xii

**TB (translation buffer)**
characteristics and use, 24, 25
entry
identified by ASN, 24
invalidating, 24
invalidating, current process, 26
invalidating, multithreaded process, 26
preventing use of stale, 26–27
invalidating
at ASN overflow, 73
SWPCTX PALcode routine, 76
**TBS (to be scheduled state)**
characteristics and use, 79
**TEB (thread environment block)**
address of, user mode thread identified by, 215
characteristics and field definitions, 213–215
layout, 214 (fig.)
**TEB$Q_SEQUENCE field**
user mode threads identified by, 215
**terminal drivers.**
job controller notified of unsolicited terminal input by, 439
unsolicited terminal input handling, 439
**terminals**
unsolicited input from, interactive processes created as a result of, 439
**termination mailbox**. *See* mailboxes—termination.
**text**
conventions, xii–xiv
**thread control block**
characteristics and use, 213
**thread environment block**. *See* TEB.
**thread manager**. *See* user mode thread manager.
**thread-safe procedures**
ACB flag that indicates, 185
AST flag that specifies, 196
criteria for, 185
effect of MULTITHREAD parameter, 185
system service flag that indicates, 188
term definition, 184

**threads of execution**
characteristics and use, 2, 151
determining limits on, within a process, 183
in a single-threaded process, 82
**TIE frame register save area**
layout, 408 (fig.)
**TIE$$SETUP_IMAGE routine ([TIE]TIE module)**
operations, 405
**TIE$A_IMAGE_BASE field**
definition and use, 394
**TIE$A_ROUTINE field**
definition and use, 394
**TIE$CALLX_LOOKUP routine ([TIE]TIELOOKU module)**
locating transfer destinations, 407
transfers from
native to translated images, operations, 409
translated to native images, operations, 410–411
**TIE$CMEXEC routine ([TIE]TIESYS module)**
operations, 412
**TIE$CMKRNL routine ([TIE]TIESYS module)**
operations, 412
**TIE$DCLCMH routine ([TIE]TIESYS module)**
operations, 412
**TIE$EMULAT_TV.EXE image**
characteristics and use, 384
**TIE$IGC routine (TIESTATE module)**
control flow, 414–415
**TIE$IGCNOTDONE error status**
returned by TIE$IGC, 415
**TIE$JMP_LOOKUP routine ([TIE]TIELOOKU module)**
locating transfer destinations, 407
**TIE$JSB_LOOKUP routine ([TIE]TIELOOKU module)**
locating transfer destinations, 407
**TIE$L_ALPHA_END field**
definition and use, 394
**TIE$L_ALPHA_START field**
definition and use, 394
**TIE$L_BOUNDARY_MAP field**
definition and use, 394
**TIE$L_FEEDBACK field**
definition and use, 395
**TIE$L_VIA_MAP field**
definition and use, 394
**TIE$NATIVE_TO_TIE routine ([TIE]TIELOOKU module)**
control flow, 409
operations, 403
**TIE$N_TO_T_CH routine ([TIE]TIEEXC module)**
fault-on-execute exception handling, operations, 413–414
transfers from native to translated images, operations, 409–410
**TIE$Q_LINK_TIME field**
definition and use, 395
**TIE$SETEXV routine ([TIE]TIESYS module)**
operations, 412
**TIE$SETUP_IMAGE routine ([TIE]TIEIMG module)**
operations, 405–406
**TIE$SHARE.EXE image**
characteristics and use, 384
**TIE$T_IMAGE_IDENT field**
definition and use, 395
**TIE$T_IMAGE_NAME field**
definition and use, 395
**TIE$T_VEST_DATE_TIME field**
definition and use, 394
**TIE$T_VEST_ID field**
definition and use, 394
**TIE$UNWIND routine ([TIE]TIESTATE module)**
operations, 413
**TIE$XXXX_LOOKUP routine ([TIE]TIE module)**
locating transfer destinations, 407–408
requesting system services from translated images, 411
**[TIE]TIE module.** *See* TIE$$SETUP_IMAGE routine; TIE$XXXX_LOOKUP routine.
**[TIE]TIEEXC module.** *See* TIE$N_TO_T_CH routine.
**[TIE]TIEIMG module.** *See* TIE$SETUP_IMAGE routine.
**[TIE]TIELOOKU module.** *See* TIE$CALLX_LOOKUP routine; TIE$JMP_LOOKUP routine; TIE$JSB_LOOKUP routine; TIE$NATIVE_TO_TIE routine.
**[TIE]TIESTATE module.** *See* TIE$IGC routine; TIE$UNWIND routine.
**[TIE]TIESYS module.** *See* TIE$CMEXEC routine; TIE$CMKRNL routine; TIE$DCLCMH routine; TIE$SETEXV routine.
**time.** *See* date; quantum; scheduling; $SETIME; timers; wait states; watch chip.
**time-of-year clock**
setting and reading, capabilities use during, 44
**timeouts.** *See also* timers—requests.
deadlock search initiation controlled by, 142
queue
deadlock handling use of lock, 142
linking locks into, 118
**timer interrupt.** *See* interval timer.
**timer queue entry.** *See* TQE.
**timers.** *See also* PCC; sanity timer mechanism; SCC.
interval. *See* interval timer.
requests
transition states triggered by, 41
software. *See* software timers.

**TIMESCHDL module**. *See* SCH$THREADQ_END routine.
EXE$TIMEOUT. *See* EXE$TIMEOUT routine.
**TMPMBX (create temporary mailbox privilege)**. *See* privileges.
**TM_SUPPORT module**. *See* EXE$CALLBACK_JACKET; EXE$CHECK_UPCALL; EXE$CREATE_KERNEL_THREAD; EXE$DELETE_KERNEL_THREAD; EXE$END_MULTITHREAD; EXE$FORCE_QUANTUM_END; EXE$INFORM_TM_AST; EXE$INFORM_TM_CALL; EXE$INIT_MULTITHREAD; EXE$KTHREAD_DELETE_CALLBACK; EXE$KTHREAD_DELETE_SELF; EXE$KTHREAD_START; EXE$REGISTER_THREAD_MANAGER; EXE$SINGLE_THREAD; EXE$SINGLE_THREAD_INT.
**TQCNT job quota**
waiting for, context, 54
**TQE (timer queue entry)**. *See also* time; timers—requests.
quota
charged for common event flag cluster creation, 85
transition states triggered by, 41
waiting for, context, 54
**traceback condition handler**. *See* condition handlers.
**transfer address**. *See* transfer address array.
**transfer address array**
characteristics and use, 360–362
field definitions, 360
image activator operations, 339
layout, 361 (fig.)
modified for multithreaded application, 212
term definition, 301
use in
multi-image process, 458, 466
single-image process, 436
**transfer vector**. *See* transfer address array.
**translated image descriptor entry**
layout, 405 (fig.)
**translated images**
activated, table describing, 405, 406
architectural support for, 389–390
bag routine, 394
function, 404
calling native images, 396
characteristics and use, 383
condition codes use by, 385
contents, 384, 392
data parameter area
follows bag routine procedure descriptor, 405
layout, 395 (fig.)
descriptor entry, layout, 405 (fig.)
distinguished from native images, 393
encapsulated VAX call frames
characteristics and use, 410
example, 411 (fig.)
executable
entry point is transfer routine, 406
initiating, 404–406
organization, 393 (fig.)
image activator operations, 347
initiating, 404–406
interpretation of VAX code, 413
operations, overview, 384
PC use by, 386
prevention of inner mode execution, 401, 402, 412
register use in, 385 (table)
requesting system services from, 411–413
run-time support for, 384
shareable
contents, 393
initiating, 406
multiple entry points, 407
organization, 394 (fig.)
replacing with native shareable image, 396
stack use by, 385
table, use by TIE$XXXX_LOOKUP to locate transfer destination, 407
TIE stack frame procedure, 408
transfers
locating destination of, 407–408
from native images, 408–410
to native images, 410–411
VEST-built data structures, 394
virtual address grain, 392
virtual address layout, 392
**translated procedures**
identifying from procedure value, 397
**$TRANSLATED_TO_NATIVE system service**
characteristics and use, 384
control flow, 403–404
**translation**
binary
advantages, 383
design goals, 383–384
**translation buffer**. *See* TB.
**translation-not-valid exception**. *See* page fault.
**TRAPB instruction**
use
by $NATIVE_TO_TRANSLATED system service, 402
in translated images, 387
**[TTDRVR]TTYCHARI module**. *See* TTY$PUTNEXTCHAR routine.
**TTY$PUTNEXTCHAR routine ([TTDRVR]TTYCHARI module)**
unsolicited terminal input handling, 439

**type-ahead buffer**
allocated at unsolicited terminal input, 439

**UAF$Q_DEF_PRIV field**
process privilege mask, use and routines that manipulate, 380 (table)
**UAF$Q_PRIV field**
process privilege mask, use and routines that manipulate, 380 (table)
**UIC (user identification code)**
determined at process creation, 271
use in locking a resource, 107
**UIC group number**
identifying common event flag cluster with, 85
lock information access restricted by, 139
process name qualified by, 225
sharing event flags among processes in a, 81
**UIC-based domain**
term definition, 472
**unaligned data**
translated image use of, 387
**unsolicited terminal input**
job controller notified of, 439
term definition, 439
**Unwind Call Chain system service.** *See* $UNWIND.
**upcall table**
allocated from nonpaged pool and P1 allocation region, 166
contents, 165
EXE$HIBER use of, 244
indicating
event flag setting, 166
thread quantum-end event, 166
locating, 166
**upcalls**
avoiding during image exit, 367
blocking, 205, 217
characteristics and use, 204–209, 243
conditions that must be met for, 205
DECthreads accepting unblocking, 217
event flag setting, 210
events that can lead to, 204
executive routines that implement, 205
exit, results of, 220
forced exit notification, when used, 253
MULTITHREAD parameter effect on, 216, 217
routines that make, 205–207
term definition, 152
use to notify DECthreads of events, 216
user mode
AST delivery, 218
thread manager actions, 47
thread quantum end, 209
when kernel thread is being waited, 180
**user identification code.** *See* UIC.
**user mode.** *See also* access modes.
logical names used in CLI initialization, 448
outermost access mode, xiii
stack pointer, as part of hardware context, 20
**user mode stack.** *See* user stack.
**user mode thread manager**
applications built with, 243
deregistering, 168–169
executive notification of, 204
forced exit notification, 253
image exit, operations, 366
placing kernel thread into hibernation, operations, 244
registering, 165–167
SCH$POSTEF notification of, 210
stalled user mode threads unblocked by, 180
upcalls
characteristics and use, 47
at event flag setting, 100, 101, 210
at event flag wait, 92, 96
waking kernel threads, operations, 245–246
**user mode threads**
blocking
after requesting $HIBER, 206
waiting for local event flags, 206
data structures that describe, 213–215
DECthreads role in managing, 2
DECthreads scheduling of, 2
DECthreads support of, overview, 211–212
identifying, 215–216
multiplexed in kernel threads, 152
quantum end, 209–210
control flow, 209
releasing, 194
scheduling, 243
example, 218, 219 (fig.)
scheduling policy, 178
TEB field that identifies, 213
unblocking
by DECthreads, 213
stalled, 180
when event flag wait is satisfied, 211
**user stack.** *See also* stack.
canonical DECthreads frame, characteristics and use, 217
created by DECthreads, 215
initial kernel thread
expandable, 162
layout, 215 (fig.)
pages, protection, 161
size specified to $INIT_MULTITHREAD system service, 164
TEB fields that delimit, 214
usable portion, 214
**user-written system services.** *See* privileged shareable images.

**valid page**. *See* memory management.
**value block**
characteristics and use, 109
contained in resource block, 109
**VAX architecture**
compared to Alpha, 386
instruction atomicity, 390–391, 414–415
memory references, compared to Alpha, 387–389
**VAX Environment Software Translator utility**. *See* VEST utility.
**vectors**. *See* condition handlers.
**VEST (VAX Environment Software Translator) utility**
atomic memory access, 388
D floating-point handling, 386
data structures built by, 394
images that cannot be translated by, 395
imprecise exception handling, 387
merging image sections, 392–393
overview, 383, 384
reproducing VAX behavior, 386
role in emulating VAX environment, 385
translating images
main phases, 391–392
requiring instruction atomicity, 391
with unaligned data, 387
**victim selection**
in deadlock handling operations, conversion deadlocks, 148–149
**virtual address space**. *See also* P0 space; P1 space; P2 space; S2 space.
characteristics and use, 4
components, 5 (fig.)
deleting during process rundown, 374
described by hierarchy of page tables, 4
managed by sections, 302–303
as part of hardware context, 20
regions, 1
shareable images, assigning, 358
**virtual block number**. *See* VBN.
**virtual memory**. *See* address space; memory management; P0 space; P1 space; pages; PTE; system space; virtual pages.
**virtual region**
characteristics and use, 289
**virtual terminals**
reconnection by LOGINOUT, 448
**VMS**
operating system names, xii
**VMScluster systems**. *See also* CLUSTER_SERVER process; locks used by OpenVMS Alpha components; resources; synchronization.
Alias locks, characteristics and use of, 497–499
all kernel threads in process execute on the same node, 228
conducting parallel remote process scans on, 229
creating a top-level process in, 265
CWPS routine
extending process control and information system services with, 225
locating processes on VMScluster nodes, 228
EPID
unique for kernel thread on, 154
used to identify node of process or kernel thread, 279
event flags not visible clusterwide, 81
$GETJPI support, 240–241
granting locks on, resource master operations, 109
identifying a process within, 224
LKB distribution in, 122
local node, resource use by, 111
membership stabilizing, RWCLU wait state use with, 39
node search use of CSID, 231
nodes, stalling execution on, wait state, 39
resources
root resources directory, 124
sharing in, 472
RSB distribution in, 122
synchronizing
resources, lock management role in, 109
time
setting coordination, 474
wildcard request handling, 232
**[VMSLIB]LIBMERGE module**. *See* LIBSP1_MERGE routine.
**Volume Allocation lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 478–479
**Volume Blocking lock**. *See also* locks used by OpenVMS Alpha components.
characteristics and use, 479–480

**Wait for Logical AND of Event Flags system service**. *See* $WFLAND.
**Wait for Logical OR of Event Flags system service**. *See* $WFLOR.
**Wait for Single Event Flag system service**. *See* $WAITFR.
**wait queue header**. *See* WQH.
**wait queues**
characteristics and use, 15–17
common event flags, 85 (fig.)
event flag cluster, CEB fields that define, 86
layout, 17 (fig.)
location, 16
lock, removing LKBs from, 139
resource, listhead location in RSB, 112
**wait states**. *See also* event flag wait state; event

flags; I/O; scheduling; states; swapper; synchronization.
CEF, characteristics and use, 16
characteristics and transitions, 35–41
context for, 52
memory management
characteristics and transitions, 36–37
context for, 53
transition to COM state, control flow, 60–63
upcall when kernel thread is placed in, 180
voluntary, transition descriptions, 35–36
**WAIT$JKT routine (SYSTEM_SERVICE_ DISPATCHER module)**
waiting for inner mode semaphore, control flow, 190–192
**$WAITFR (Wait for Single Event Flag system service)**
kernel thread state change actions, 32, 35
program flow example, 94, 95(fig.)
requesting event flag wait, control flow, 91
**$WAKE (Wake system service)**
operations, 245–246
process state change actions, 35
**Wake system service.** *See* $WAKE.
**WCB (window control block)**
KFE field that specifies, 320
**$WFLAND (Wait for Logical AND of Event Flags system service)**
AST queuing constraints on use of, 103
compared with $WFLOR, 91
kernel thread state change actions, 35
requesting event flag wait, control flow, 91
**$WFLOR (Wait for Logical OR of Event Flags system service)**
compared with $WFLAND, 91
kernel thread state change actions, 35
requesting event flag wait, control flow, 91
**wildcards**
$GETJPI support of, 242–243
indicator, EPID field negative value used as, 225
processing
$PROCESS_SCAN distinguished from $GETJPI, 229
requests, VMScluster system handling, 232
**window control block.** *See* WCB.
**word index**
term definition, xiv
**working set.** *See also* automatic working set limit adjustment; memory management.
characteristics and use, 182–183
initializing, 292
moving pages into and out of, 182
preventing changes in multithreaded process, 183
quota
restrictions enforced by EXE$PROCSTRT, 292
trimming, transitions triggered by, 36
**working set list.** *See also* memory management; page faults; swapper; WSLE.
address in CTL$GL_WSL cell, 289
keeping a page in, 97
kernel stack pages locked in, 171
permanent PKTA URUW page locked in, 165
PKTA UREW page locked in, 171
**working set list entry.** *See* WSLE.
**working set swapper.** *See* swapper.
**WORLD (affect other processes in world privilege).** *See also* privileges.
lock information access permitted by, 139
required by EXE$NAM_TO_PCB, 227
required for
implicit affinity modification, 254
process affinity modification, 254
process capabilities modification, 255
**WQH (wait queue header)**
definition and use, 15–17
layout, 17(fig.)
**WQH$L_WQCNT field**
kernel thread state wait queue use of, 16
use by SCH$REPORT_EVENT, 62
**WQH$L_WQSTATE field**
kernel thread state wait queue use of, 15
**WRITE_UNQ PALcode routine**
operations, 217
**WSLE (working set list entry)**
dynamic, PHD field that contains number of, 373
**WSMAX parameter (SYSGEN)**
use during working set initialization, 290, 292

**XDELTA.** *Consult* OpenVMS Delta/XDelta Debugger Manual.
**XQP (Files-11 Extended QIO Processor).** *See also* file system.
lock use by, characteristics and use, 478–485
per-process initialization, 295
and process deletion, 422
system-owned locks use by, 136

**zeroed page list**
supplied by SCH$IDLE, 77
**ZERO_LIST_HI parameter (SYSGEN)**
maximum number of zeroed free pages, 76

VERSION 7.0

# OpenVMS Alpha Internals

## SCHEDULING AND PROCESS CONTROL

RUTH E. GOLDENBERG • DENISE E. DUMAS • SARO SARAVANAN

***OpenVMS Alpha Internals: Scheduling and Process Control*** provides an update to selected parts of the book *OpenVMS AXP Internals and Data Structures Version 1.5*. (Digital Press, 1994). This book covers the new scheduling model in OpenVMS Alpha Version 7.0, which includes executive support for multithreading. It also discusses the life of a process, from creation to deletion.

This book is intended for systems programmers, technical consultants, application designers, and other computer professionals interested in learning the details of the OpenVMS executive. Teachers and students of graduate and advanced undergraduate courses in operating systems will find this book a valuable study in how theory and practice are resolved in a complex commercial operating system.

**Ruth E. Goldenberg** is a consulting software engineer for the OpenVMS operating system development group at Digital Equipment Corporation and is the co-author of *OpenVMS AXP Internals and Data Structures Version 1.5* and *VAX/VMS Internals and Data Structures Version 5.2*.

**Denise E. Dumas** is consulting software engineer for the OpenVMS operating system development group at Digital Equipment Corporation. She is a co-author of *VAX/VMS Internals and Data Structures Version 5.2*.

**Saro Saravanan** is an independent consultant. He is the co-author of *OpenVMS AXP Internals and Data Structures Version 1.5*.

dp Digital Press
An imprint of Butterworth-Heinemann